Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2012-033 October 4, 2012

Monitoring the Execution of Temporal Plans for Robotic Systems

Steven J. Levine

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Monitoring the Execution of Temporal Plans for Robotic Systems

by

Steven James Levine

S.B., Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

August 2012

Author .....
Department of Electrical Engineering and Computer Science
August 17, 2012

Certified by .....
Brian C. Williams
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by .....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Monitoring the Execution of Temporal Plans for Robotic Systems

by

Steven James Levine

Submitted to the Department of Electrical Engineering and Computer Science
on August 17, 2012, in partial fulfillment of the
requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

To achieve robustness in dynamic and uncertain environments, robotic systems must monitor the progress of their plans during execution. This thesis develops a plan executive called Pike that is capable of executing and monitoring plans. The execution monitor at its core quickly and efficiently detects relevant disturbances that threaten future actions in the plan. We present a set of novel offline algorithms that extract sets of candidate causal links from temporally-flexible plans. A second set of algorithms uses these causal links to monitor the execution online and detect problems with low latency. We additionally introduce the TBurton executive, a system capable of robustly meeting a user's high-level goals through the combined use of Pike and a temporal generative planner. An innovative voice-commanded robot is demonstrated in hardware and simulation that robustly meets high level goals and verbalizes any causes of failure using the execution monitor.[1]

Thesis Supervisor: Brian C. Williams
Title: Professor of Aeronautics and Astronautics

[1]This research is generously funded by the Boeing Company grant MIT-BA-GTA-1.

Acknowledgments

Wow, my thesis is finally done! There is absolutely no way that I could have gotten here without my friends and family, so I would like to thank them in this section.

I'd like to thank my lab, the MERS group in CSAIL, for their constant support over the course of my work. Professor Brian Williams introduced me to the intricacies of model-based robotics and provided research guidance and writing advice throughout this thesis process. I appreciate that he genuinely cares about the quality of our work, and he strives to push us to become the best researchers that we can be. I'd also like to thank all of my labmates. Pedro Santana, whom I worked very closely with in developing our robot under the Boeing manufacturing project, brightened up some of the most stressful days with his fantastic sense of humor. Also, the vision-based sensing system that he built for our robot is actually quite exceptional (despite what I might say when things aren't working... :-). David Wang took me under his wing when I first joined the group and spent a lot of his personal time teaching me how things work in our group as well as implementing some key algorithms for our robot. Andreas Hofmann provided very useful guidance when we were building our robotic testbed, and always has great stories to share over a slice of pizza. I'd like to thank Peng Yu for his stable work style and his willingness to take a break and chat about research ideas, computer hardware, and cameras. Andrew Wang is a great friend of mine, and I am lucky enough to sit across from him after having met him five years ago at the beginning of freshman orientation. I'd like to thank Wesley Graybill for his bicycle chat and helping me layout my thesis, Eric Timmons for bringing cool hardware ideas to lab, Simon Fang for forcing me to take a break and grab a cup of coffee when I need it most, Larry for providing comic relief to our group, Shannon for first teaching me how to use the WAM arms, Hiro Ono for making the lab more musical, and Bobby Effinger for some interesting conversations. I'd also like to thank Scott Smith and Ron Provine from the Boeing company, for their endless supply of interesting research ideas to pursue at our biweekly meetings. Also in CSAIL, I'd like to give a big "thank you" to Ron Wiken, the humble man who always has ingenious

save-the-day project advice and keeps the hardware in CSAIL running. One day I'll finally find something he doesn't have in that great stockroom of his.

At MIT, there have been a number of people who have had a profound influence on me. I must thank Paul Gray for being an excellent and understanding faculty advisor, Wayne B. Johnson for really caring about us advisees even after we became upperclassman, and many faculty friends from LeaderShape including Jed Wartman, Tracey Purinton, Kirk Kolenbrander, John DiFava, and Victor Grau Serrat.

There's no way I could have gotten here without a lot of help from my friends. Frank Yaul, my great roommate, helped keep me sane at some points when things weren't working out (both in life and in research). Thanks for always being there to listen, and for some great biking adventures (and many more to come)! Scott Vasquez, my freshman year roommate - thank you for stopping by lab this year to "keep it real" and get me to take a break once in a while. Thank you to Wei-Yang Sun, a good friend of mine from undergrad, for those great musical jam sessions. And of course, all of my great friends from Next House! To the Next-Make crew - Marcel Thomas (my other roommate!), Scott Bezek, Feng Wu, and Joe Colosimo - the system we built was the epitome of my senior year, if not my entire undergraduate career. Thanks for all of the great times, and I can't wait to do more fun projects with you guys in the future. Thank you to the Next 5W crew for innumerable great memories hanging out in the lounge late into the night doing problem sets. Thank you to all of my LeaderShape friends - it was a life-changing experience (both times!). Thank you to all of my high school friends - particularly Christa Levesque and Chris Wong - for providing a much-needed escape from the MIT bubble once in a while.

I've had some excellent, genuinely passionate teachers while going through the Framingham public school system. Looking back, I now realize just how special a place Framingham is. My teachers have changed my life, and I'd like to thank them in this section. Thank you Ms. Barstow, for being a great role model and for teaching me how to "get organized" - a skill I sometimes forget but am still working on. :-) Thank you to Dr. Langdon for his passion and invaluable mentorship during my freshman year biology class. If it weren't for that science fair, I would not be here at

MIT today, embarking on a journey to pursue my Ph.D. Thank you Mr. Coleman, my superb calculus teacher, for your dedication to teaching and for letting me try my software in your classroom. And of course, thank you Mr. Witherow for a great year of physics and for genuinely caring about our success and future education.

My family has shaped who I am today, and I am deeply indebted to them. I would like to thank my loving grandparents - Papa Al, Grandma Marcia, Grammy Elaine, Auntie Laura and Uncle Dave, Auntie Cindy and Uncle Mark, Shari and Lori, Uncle Eric and Aunt Debbie, and Melissa and Jessey, and of course my little white, fluffy friend Dewey. Even though Grampy, Papa Eddie, and Grandma Marie aren't here on this day, I know that they are watching from above. I have so many warm memories growing up in this family, and I can't begin to thank them enough for their constant love and support over the years.

Lastly, and most importantly of all, I would like to thank my parents, Leslie and Mark Levine. They have been, without a doubt, the strongest influences in my life. Their unconditional love has guided me and molded the core values and beliefs that make me who I am today. They have given me life, taught me how to live, given me advice when I need it most, and always thoughtfully listened whenever I had a problem (no matter how trivial). There is absolutely no way that I would be here if it weren't for their selfless love. No words in the English language are strong enough to express my gratitude towards them. Thank you so much Mom and Dad for everything - I love you both very much.

Contents

List of Figures

Chapter 1

Introduction

It has long been a hallmark of artificial intelligence to develop robotic systems that accept user-commanded goals at a high level, generate and execute plans designed to meet those goals, and if necessary generate new plans that overcome any unexpected problems that arise during execution. This thesis works towards that goal, placing special focus on the subproblem of quickly detecting unexpected problems.

Specifically, this thesis is concerned with the advancement of robotic systems that are capable of robustly recovering from disturbances via model-based reasoning over plans and environmental observations. We envision robots that are highly adept at seamlessly detecting and recovering from failures in their plans early and proactively, before they become catastrophic later on. We argue that monitoring the online execution of plans is crucial for achieving robustness, and hence for building robots that seem to act intelligently in their environment. Chapter 2 introduces the core technical contribution of this thesis, which is a set of algorithms for automatically extracting causal rationale from temporally flexible plans. This information is used in a set of online plan execution algorithms to detect unexpected disturbances that jeopardize future actions in the plan with very low latency. In Chapter 3, we introduce a system that re-plans once these disturbances are detected in order to meet a user's high-level goals. We combine the merits of temporal generative planning with plan execution and monitoring. In Chapter 4, we demonstrate an implementation of a robot that is capable of achieving robustness in the face of unexpected disturbances

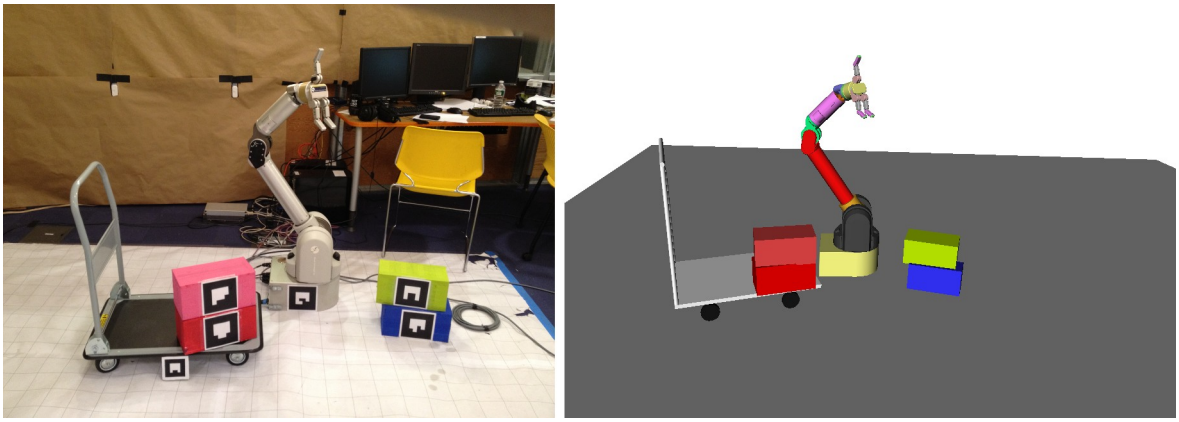

Figure 1-1: Our voice interactive robot using the technologies we develop. The left shows the actual robot in our testbed, a block stacking WAM arm. At right is our simulation environment. The user can talk to this robot and give it goals verbally, which it carries out in the face of disturbances and moving blocks. When something unexpected happens, the robot explains the cause of failure and why it is a problem with respect to its current plan.

while simultaneously interacting verbally with the user.

1.1 Robotics Scenario

We first briefly introduce the end result of the system we develop - a voice interactive robot capable of robustly meeting user goals within the context of block stacking. Please see Figure 1-1. This system employs a task executive complete with execution monitoring and generative planning. A user is able to specify a desired block configuration by verbally interacting with the robot. For example, a person may say "Make pallet," after which the robot will proceed to make the desired tower. Disturbances about in this scenario - the robot may accidentally drop blocks, a human may come and move blocks (either helping or hindering the plan), etc. The robot is able to immediately detect all of these disturbances and produce new plans to correct them and meet the users original goals. When one of these disturbances occurs and replanning is required, the robot provides intuitive explanations for the problem with such phrases as "I can't stack the pink block on the red block because the red block isn't free above."

This thesis describes the core components that make the above robot possible. We briefly highlight this subcomponents in the following sections.

1.2 Plan Execution and Monitoring

In Chapter 2 of this thesis, we present Pike - a system for plan execution and monitoring. Execution monitoring is crucial for developing robotic systems that can robustly recover from disturbances. It is necessary to make use of observed data about the world state in order to judge whether a plan being executed is evolving correctly or not. The job of execution monitoring is to detect problems in a plan early, before they escalate and compound to much larger problems later in the plan. By detecting relevant disturbances early, we argue that robotic systems will have greater flexibility in replanning and will thus be considerably more robust.

Our approach to constructing the Pike plan executive is to divide the task of plan execution into two steps: 1.) that of temporal scheduling, dispatching, and monitoring, and 2.) that of monitoring state constraints of the actions in the plan. This thesis focuses mainly on the second point, and is what we refer to as execution monitoring.

Our approach introduces two sets of algorithms: an offline set responsible for extracting the cause or rationale behind actions in the plan (called causal links), and an online set that is responsible for using this information to monitor relevant conditions in the world during plan execution. A key novel contribution of this thesis is the automatic extraction of sets of candidate causal links for plans with temporal flexibility. Prior work has generally assumed causal links can be obtained as a side effect of the planning process, or can be extracted from totally-ordered plans ([29], [24], [20], [8], [19]). We wish however to provide a general framework capable of extracting causal links from plans generated by any temporal planner, and hence can make no such assumptions. Such a framework would be of use to the planning community, as it would effectively allow any temporal generative planner to be function as a continuous planning and execution system. We present the algorithms

for causal link extraction and monitoring in this chapter, as well as provide proofs and correctness guarantees. We further provide empirical simulation results that show very low latencies in detecting violated plans (around 20-80 milliseconds).

1.3 The TBurton Executive

Chapter 3 of this thesis introduces the TBurton executive, a system that combines the plan execution proficiencies of Pike with a temporal generative planning algorithm in order to achieve robustness with respect to user-specified goals in the face of disturbances. Intuitively, the TBurton executive employs re-planning to chart out a new course of action when the execution monitoring component of Pike signals that a relevant disturbance has occurred that threatens the current plan. By replanning at fast time-scales, we are capable of achieving robotic systems that are persistent at reaching a user's goal, and are successful under some mild assumptions.

1.4 Innovative Application: Voice-Interactive Robotic Manufacturing

Chapter 4 of this thesis applies the ideas of the first two chapters and describes the innovative application noted above. We demonstrate a robot, implemented in hardware and in simulation, that is capable of robustly achieving user goals specified at a high level and explaining any causes of failures, all within the limited context of robotic manufacturing. This robotic system can be commanded verbally, and is able to additionally explain the reason why a disturbance is relevant at the plan level. This chapter describes in detail the key features and implementation of our robotic system.

Chapter 2

Plan Execution and Monitoring

Execution monitoring is the problem of verifying that a robotic system is executing a task correctly, and signaling an error as early as possible should a disturbance be detected. Classical planning deals only with the problem of generating high-level plans for a system. However, the continuous planning and execution that is required for deliberative, robust task execution requires an execution monitoring capability to detect the unexpected.

Our approach to execution monitoring is based on the use of causal links. While previous work has explored using causal links to generate plans ([24], [20]) and subsequently using these planner-generated causal links for online monitoring ([29], [8], [19]), our approach is novel in that it extracts sets of causal links from already-existing *least-commitment plans*, allowing plans with temporal flexibility to be effectively monitored independent of the planning algorithm used to generate them. A key contribution of this thesis is the set of algorithms that automatically extract causal links from temporally flexible plans. By working with least-commitment plans that include temporal flexibility as opposed to inflexible points with rigid time requirements, our robotic systems will be less brittle and more realistic. An additional benefit of our approach is that it is independent of the planning algorithm that produced the plans, and hence can be easily plugged into related systems that use different planning algorithms.

This chapter develops a plan executive for executing and monitoring temporal

plans on robotic system. Our main focus will be on the execution monitoring component. We hence begin by motivating the need for execution monitoring in real robotic systems. We then proceed to introduce certain necessary prerequisites to the plan execution problem - namely action planning and temporal plans - allowing us to formalize a problem statement. Then, we proceed to describe our algorithms for plan execution and monitoring, which consist of an offline causal link preprocessing phase as well as an online dispatching and monitoring phase. Proofs of correctness are included, as well as a theoretical asymptotic performance analysis. We then conclude this chapter with experimental test results.

2.1 Motivation and Goals

In this section, we motivate the need for and importance of execution monitoring. We begin by discussing some of the key assumptions used in classical planning, and discuss why some of these assumptions are not realistic for real-world task execution.

2.1.1 Assumptions used in Classical Planning

Task execution is very different in a highly-dynamic environment than in the clean, deterministic setting assumed in classical planning. In particular, traditional classical planners make several implicit assumptions about how their plans will be executed that do not necessarily hold in the real world:

- **Sole agent of change.** Classical planners consider only static environments, where the environment changes only through deliberate actions by a single agent. In other words, properties of the environment only change when they are modified by the agent. However, this assumption does not hold in many real-world situations, particularly those that are dynamic, uncertain, or uncontrollable. Properties and objects in these sorts of environments can change spontaneously and without warning, possibly in disruptive ways that can cause a preconceived plan to fail.

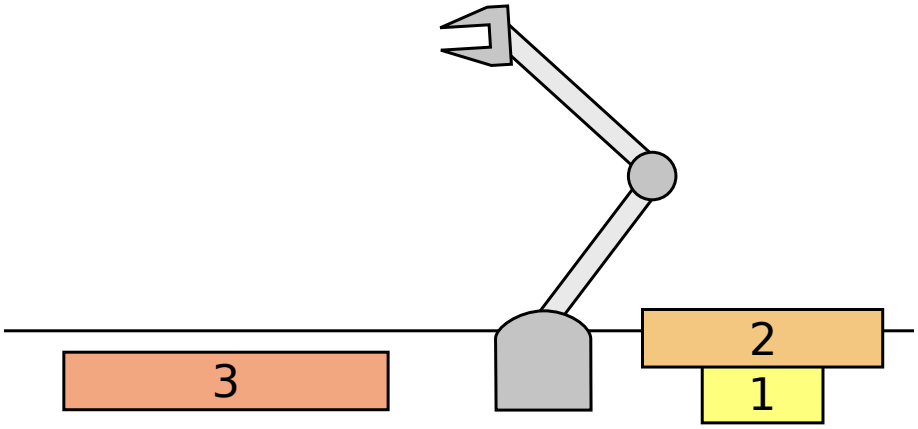

Figure 2-1: An idealized rendition of the classic BlocksWorld domain. The robotic gripper can move around to pick up and put down a single block at a time. The goal here is to stack the pyramid "1, 2, 3" on the ground. In this depiction, there are no sources of uncertainty, and similarly there will be no disturbances to the system. This is representative of the nominal environment used by traditional classical planners.

- **Instantaneous actions and state transitions.** Traditional classical planning has no notion of time. Actions are assumed to have no duration, and discrete state transitions happen instantly.

We now present an example scenario that makes use of these assumptions. Consider the well-known BlocksWorld domain, illustrated in Figure 2-1. In this domain, a robot with a gripper is tasked with manipulating blocks. The gripper is capable of picking up blocks that are either on the ground or on top of other blocks, and also putting blocks down (again, either on the ground or on top of other blocks). In the illustration, the robot's goal is to create a pyramidal stack. A typical classical planner will generate a plan like: 1.) Pick up Block 2, 2.) Stack Block 2 on top of Block 3, 3.) Pick up Block 1 from the ground, 4.) Stack Block 1 on top of Block 2.

The BlocksWorld domain makes the assumptions noted above, which do not necessarily hold for real-world task execution. Specifically, BlocksWorld assumes a completely deterministic environment in which the robot is the sole agent of change. No blocks will spontaneously move by themselves or fall over. There are no unmodeled agents that may move blocks (either malicious or benevolent) without the planner

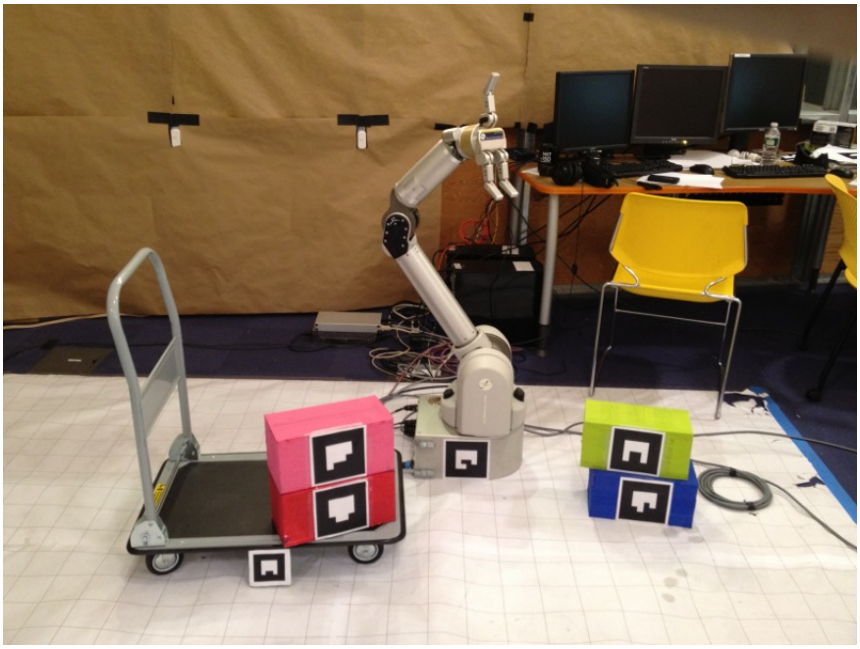

Figure 2-2: A real-world and hence much "messier" implementation of the BlocksWorld domain. Many of the assumptions made by classical planners about the world in which their tasks will execute are not valid. For example, actions may fail, blocks may fall, additional autonomous agents (either malicious or benevolent) may unexpectedly change the world state, etc.

telling them to do so. Traditional classical planners assume that, should the robotic agent place Block 2 on top of Block 3, that it will stay there indefinitely unless the robot proceeds to manipulate it in the future.

While very useful from a planning perspective, these assumptions are not valid in real-world execution environments. For example, consider the much more realistic implementation of BlocksWorld depicted in Figure 2-2. When executing in the real world with faulty hardware and uncertain sensor data, it is quite possible (and even likely) that actions will fail. A command generated by the planner such as "stack Block 2 on top of Block 3" may very well result in any of the following actually happening: the robot placing Block 2 on top of Block 3 (normal behavior), the robot placing Block 2 on on the edge of Block 3 immediately after which it falls off, the robot accidentally dropping Block 2 in transit, the robot accidentally knocking over Block 3's tower while approaching with Block 2, a small nearby toddler grabbing the

block from the robot's gripper, or any number of other unexpected activities. In short, the real world can be characterized by the existence of unexpected and oftentimes unmodeled disturbances.

The clean world for which classical planners generate their plans is perfectly reliable and deterministic. Real world robotic systems that make these assumptions are however often somewhat brittle in practice; it is usually difficult to guarantee the actual outcome of dispatching a command to a robot. Systems that operate open-loop without any feedback often succeed only in nominal environments where no disturbances occur. Continuing with the example above, suppose that the action "stack block B on top of block C" fails because the robot accidentally drops the block on the ground while in transit. If the system were to naively continue on to the next step in the plan regardless of the failure, the robot would pick up Block 1 and try to stack it on Block 2 (which is on the ground). This action will also therefore fail, since Block 2 is not where the open-loop task executive expected it to be. Failures will compound; a single problem can cascade and cause the entire plan to fail.

Therefore, in dynamic and unstructured environments, it is necessary to detect in real-time those disturbances which will prevent an executing task from completing successfully. Once these disturbances are detected, a high-level planner can subsequently resolve the issue by generating a contingency that circumvents the exception condition. This is the goal of execution monitoring - to detect in real time when relevant disturbances to the plan occur. Execution monitoring is therefore necessary for building robotic systems that operate robustly in practice.

2.1.2 Need for Immediate Fault Detection

In the previous section, we motivated the need for monitoring by noting some of the assumptions made by classical planners and describing how they do not hold in systems operating in the world. Specifically, the robot is often not the sole agent of change since disturbances can occur spontaneously outside of the robot's control. In this section, we aim to motivate the need to detect these disturbances as early as possible, as opposed to lazily detecting them later.

Intuitively, the key idea of this section is that by detecting a potential problem early, we maximize our flexibility in replanning. There will likely be more contingency options available in this case. On the other hand, if we wait until later, we may waste time taking remedial actions that could have been avoided had the problem been dealt with earlier, even though the disturbances may have been simpler to detect.

Consider the two plausible monitoring schemes, both of which check to see if an action if capable of running [25]:

- **Action Monitoring.** Immediately before executing each action in a plan, check to make sure that the action can run correctly. If it can, run it. If not, make a new plan.

- **Execution Monitoring.** Whenever something changes in the world, check to see if it will make any action later in the plan unable to run. If not, then ignore this change. If so, then make a new plan.

Though both of the schemes appear similar at first glance and will prevent a system from executing invalid actions (such as stacking Block 1 on Block 2 if block isn't there), they result in qualitatively different behavior. Action monitoring lazily checks that the preconditions necessary for each action to run only immediately before running that action, whereas execution monitoring looks deep into the plan whenever anything in the world changes to see if it will be a problem later on. To clearly illustrate the difference between these two strategies and demonstrate why immediate detection is crucial, we once again embrace our beloved block stacking robot as an example. This time, the robot is asked,

"Friendly robot, could you please build a pyramid in 20 seconds?"

The results of this timed-goal are illustrated in Figure 2-3 for Action Monitoring and 2-4 for Execution Monitoring, as well as summarized in 2-5.

The robot begins by stacking Block 2 on top of Block 3 in both monitoring cases. However, at time $t = 6.5$ we introduce an unexpected disturbance into the system. Block 2 falls off of Block 3 and lands nearby on the ground. This is where Action

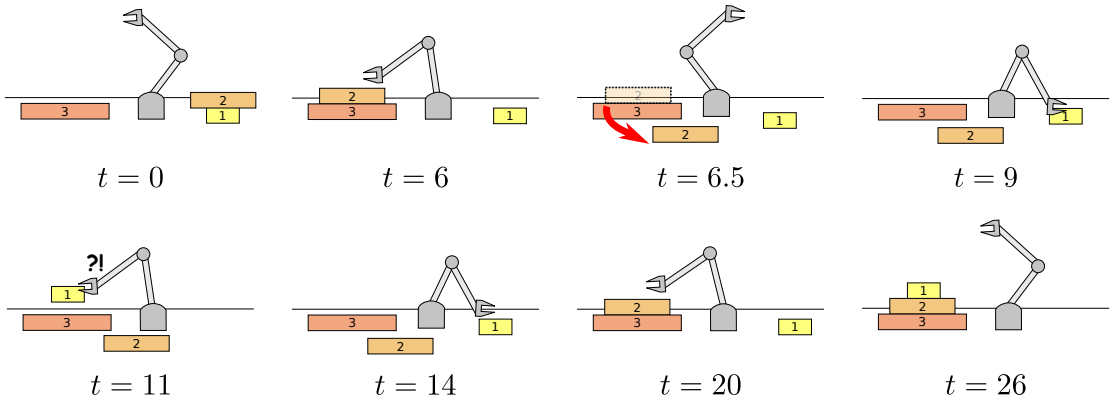

Figure 2-3: Action Monitoring. The block stacking robot using action monitoring instead of execution monitoring. The robot lazily checks if actions are valid only right before executing them. The robot begins at $t = 0$, tasked with stacking the three blocks into a pyramid. It continues successfully through $t = 6$, completing Block 2. However at $t = 6.5$, a disturbance is introduced. The robot doesn't care however because the disturbance isn't a problem yet; Block 2 doesn't affect the next action (which is to pick up Block 1). Hence the robot continues on through $t = 9$, obliviously picking up Block 1. It does not care about the problem until $t = 11$, when it is about to put Block 1 on top of where Block 2 used to be, but Block 2 is not present. Hence, to recover from the problem, the robot must take remedial action and put down the block it's holding ($t = 14$) before continuing. It succesfully puts Block 2 back on top of Block 3 by $t = 20$ and finishes the task at $t = 26$.

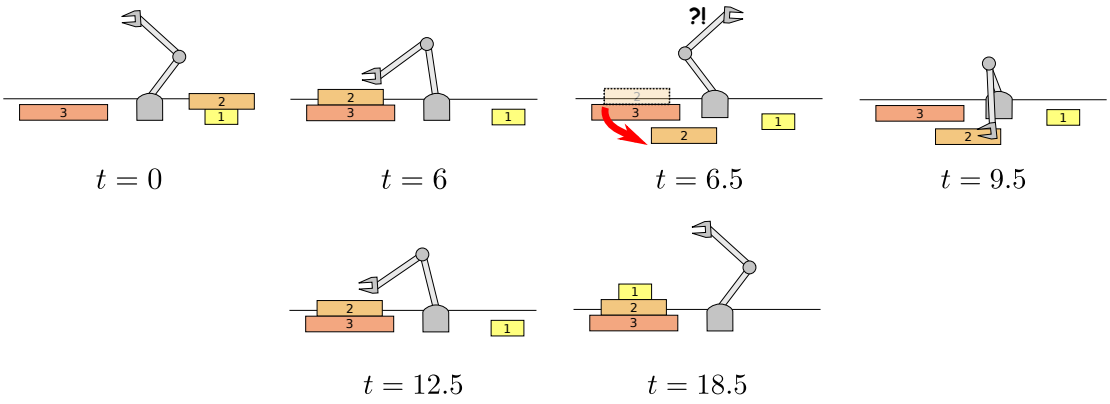

Figure 2-4: Execution Monitoring. Our beloved block-stacking robot is now using execution monitoring instead of action monitoring. It notices relevant disturbances immediately without waiting until the disturbance actually becomes a problem. The bot begins at $t = 0$, and accomplishes stacking Block 2 on top of Block 3 by time $t = 6$. Again at $t = 6.5$, a disturbance is introduced. With execution monitoring, the robot notices the problem immediately (as opposed to execution monitoring, where a fault isn't detected until a few seconds later). Realizing that the disturbance will threaten a future step in the plan (namely, stacking Block 1), the robot takes corrective action now while it is still easy. At $t = 9.5$, our bot picks up fallen Block 2 and has restacked it on Block 3 by $t = 12.5$. It's clear sailing from there to the goal, which is completed at $t = 18.5$, 7.5 seconds earlier than execution monitoring's completion time.

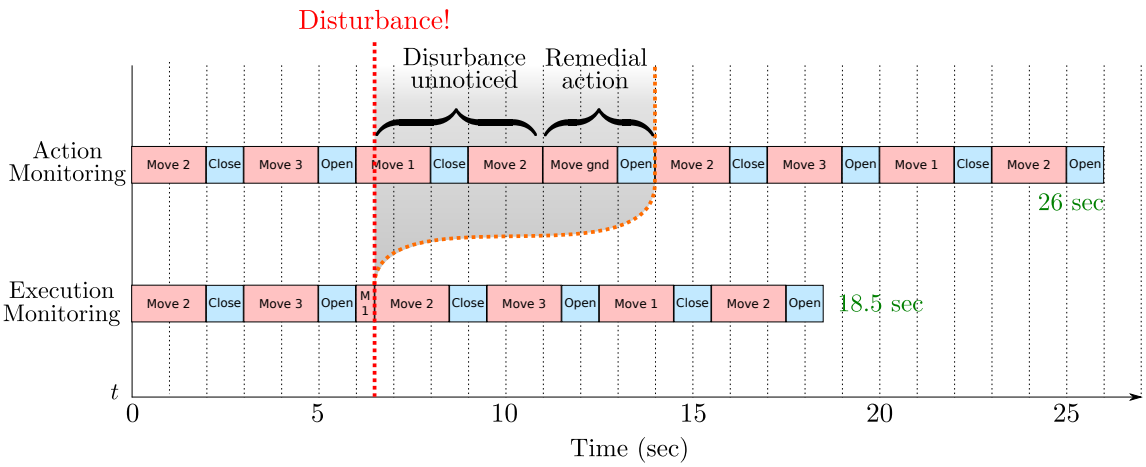

Figure 2-5: A performance comparison of action monitoring versus execution monitoring for a task with a disturbance. Each red, move-to or move-above action takes 2 seconds, and the blue gripper actions each take 1 second. The red dotted line denotes the disturbance of moving Block 2 to the ground. Action monitoring results in suboptimal execution, which are shown in the gray shaded region between the red and orange dotted lines. Execution monitoring completely avoids these unncessary actions, and hence is able to finish 7.5 seconds faster and completely the goal successfully.

Monitoring and Execution Monitoring diverge. Action monitoring does not care about this disturbance at the time when it happens, because it is engrossed with picking up Block 1 (of which the fallen Block 2 is completely irrelevant). It is not until later (namely, $t = 11$ when the bot tries to put the now-acquired Block 1 on top of Block 2) that it starts caring about the fallen block. By this point however, the situation is more complicated and the available contingency options are more costly. Since the bot is currently holding Block 1, it has to put this down before it can continue on and re-stack Block 2 on top of Block 3. It finishes the pyramid in 26 seconds, thereby not meeting the user's temporal requirement.

With execution monitoring however, the robot is able to detect immediately that the fallen Block 2 is a problem. Instead of postponing resolution, the robot selects a less-costly contingency option since it is not burdened with holding a block as was the case with Action Monitoring. The robot is able to immediately correct the problem by restacking Block 2 onto Block 3, and then proceeding to stack Block 1 on top. As

a direct result of it's immediate response and better contingency selection, the robot with execution monitoring can complete the task in 18.5 seconds, meeting the user's temporal requirement. A detailed comparison of the actions taken by the robot in each case is shown in Figure 2-5.

The crucial idea we wish to illustrate is that proactive approaches that detect and resolve problems early maximize available contingency options and can allow robots to meet goal constraints that may otherwise be unachievable if lazy monitoring schemes are used. The remainder of this chapter is devoted to developing an execution monitoring system that uses this proactive approach to detecting disturbances.

2.1.3 Goals for our Execution Monitoring System

The previous sections motivated the need for fast execution monitoring. Here, we boil down the salient features from these examples into a set of desired capabilities for our execution monitoring system:

1. **Detect disturbances as early as possible.** By detecting problems early, we maximize the chance of finding a new plan that resolves the issues. Detecting disturbances before they actually inhibit a robot from proceeding in its task may allow for more efficient contingency plans to be found.

2. **Detect only those disturbances which are relevant.** The environments in which robots operate is oftentimes very large and complex. It is therefore very useful to have guidance regarding which properties in the world are relevant to a given robotic plan. This can allow robots with limited sensing resources to focus their efforts most effectively by monitoring only properties which are relevant to the system.

3. **Operate on plans that include flexibility.** As noted earlier, it is very beneficial for generated plans to contain built-in flexibility. We therefore require that our execution monitor be able to operate with temporally flexible plans.

2.2 Prior Work

The basis for many of the ideas presented in this thesis come from prior work. In this section, we briefly describe some of these past ideas. We additionally note other contributions to the field of plan execution and monitoring that have a degree of similarity to this thesis, noting our key similarities and differences.

Perhaps the earliest robotic system to extensively use execution monitoring was the PLANEX system, which controlled a robot named Shakey at the Stanford Research Institute [10]. Their STRIPS-based system used the notion of triangle tables to monitor the execution of totally-ordered plans online as they were executing. A triangle table is a tabular array generated during planning that captures information about the the state of the world at different points during execution. During execution, the actual state of the world can be cross-referenced to this table, allowing the system to detect unexpected operator failures or serendipitous conditions in which certain actions in the plan may no longer be necessary and can be skipped. The Shakey robot used integrated planning and monitoring to alter the current plan such that the shortest executable action to reach the goal would be selected for execution. If no suitable action can be found, then replanning occurs. The execution monitoring system discussed in this thesis is similar to that used by PLANEX in that it is capable of discerning which world state changes are relevant to the world. Our approach is different in that we consider temporally flexible plans rather than totally ordered plans with no notion of time.

The SNLP planning system makes extensive use of causal links during the STRIPS planning process [20]. SNLP is a nonlinear planner, meaning that it reasons over partially ordered plans rather than totally ordered plans. SNLP is based off of the NONLIN planner ([28]), and solves the planning problem using causal links. SNLP deals with threats to causal links by adding ordering constraints to the partially-ordered plan being constructed. Much of the ground work for causal links in this thesis builds upon the definitions presented in [20]. Another planning algorithm that also makes use of causal links during the planning process is UCPOP [24]. The

UCPOP planner is able to plan over a superset of STRIPS with universally quanitifed preconditions, effects, and goals. While neither of these planners can be considered executives since they do not address plan execution, we mention them nonetheless due to their use of causal links.

Seminal work regarding the online dispatch of temporally-flexible plans is presented in [22]. This thesis makes extensive use of these techniques, and builds off of them to additionally provide facilities for state monitoring in addition to temporal monitoring. The approach taken in [22] is to execute a temporal plan by first making explicit all implicit temporal constraints via running an all-pairs shortest path algorithm. Subsequently, unnecessary constraints are pruned, leaving a minimal dispatchable network formulation that can be dispatched quickly without traversing the entire network.

Veloso et al. introduce the notion of rationale-based monitors, which are similar in spirit to the causal links used in this thesis for execution monitoring ([29]). Like causal links, rationale-based monitors capture a set of conditions about the world that are relevant to plan success, thereby allowing spurious or irrelevant conditions and changes in the world to be ignored. The approach taken in [29] generates rationale-based monitors during the planning process, which is unified with execution process, to construct a continuous planning and execution system. By extracting rationale-based monitors not only for plan conditions but also for alternative planning options that were not chosen during the planning process, plans may be quickly updated when rationale-based monitors fire. For example, it may be possible to quickly change focus and use an originally-not-chosen alternative should conditions become favorable as dictated by fired rationale-based monitors. This thesis is similar to [29] in that we also monitor only relevant conditions in the plans, with the end goal of providing a general framework for a continuous planning and execution system. Unlike that of [29], our approach is specifically interested in the explicit model of time in temporal plans and in the extraction of causal links independent of the planning algorithm used.

The Remote Agent system provides an executive capable of generating and mon-

itoring temporal plans via model-based reasoning ([23]). It was implemented and tested on NASA's Deep Space One. Remote Agent employs a mode identification and reconfiguration unit called Livingston that was responsible for monitoring the state of the spacecraft by processing low-level sensor information, and reconfiguring the spacecraft should a fault be detected.

Most prior work has considered the problem of execution monitoring plans in order to ensure that they execute correctly. Some however have also considered the problem of monitoring plan optimality [14]. This work brings up the interesting idea that, although many planning algorithms generate optimal plans, they in practice run suboptimally when dispatched in the real world. It is therefore desirable to annotate plans with conditions that can be continuously checked during execution time to query plan optimality. Since disturbances that are irrelevant with respect to optimality need not cause alarm, replanning can be minimized while simultaneously ensuring plan optimality.

The Drake system is capable of efficiently executing temporal plans with choice [3]. Temporal plans are compiled offline via labeled distance graphs. These labeled distance graphs can be dispatched efficiently online, and provide robustness at quickly making choices in the plan based on monitoring the results of the monitored schedule during execution. This can be viewed as a type of execution monitoring, as the result of sensing timing events causes Drake to react quickly and choose a different path through the plan. The Drake system is primarily concerned with making choices between contingency options with low latency at run time based on temporal constraints, not based on state constraints.

Execution monitoring has also been used to dynamically re-order actions in partially-ordered plans [21]. This approach constructs a policy via a plan regression that enables actions in these partially-ordered plans to be re-ordered based on state and action constraints, thereby improving robustness with respect to the monitoring of sequentially ordered plans. The work in this thesis is primarily concerned with monitoring temporal plans, which can be considered a super set of partially ordered plans.

Several past approaches have examined various forms of temporal logic for use in

execution monitoring systems. In [1], a temporal logic called Eagle is developed that is a superset of linear temporal logic. Their system is demonstrated on an experimental interplanetary rover in a scenario involving the detection of temporal inconsistencies in durative-action plans expressed with linear temporal logic. Our problem formulation differs in that we are also concerned with monitoring state constraints in addition to temporal constraints. These state constraints play a central role in the extraction of our causal links.

Another interesting prior approach is based on a formalism called TAL, or Temporal Action Logic ([8], [18]). This work associates monitor conditions with each action, allowing progress to be monitored. The expressive representation used allows such operator-specific monitors as "when executing pick-up-box, the box must be picked up within 2 seconds and cannot be dropped until the action completes." Granularity within each action is therefore achieved. Conditions can be constructed that monitor preconditions, prevail conditions (very similar to our maintenance conditions), effects, temporal constraints with set-bounded time invervals, and causal links between actions. Our works differ however in the manner in which these monitor conditions are generated. The approach taken in ([8], [18]) assumes that these conditions are generated automatically from the planning process (namely, the TALPlanner presented must generate all causal link monitor conditions). We however take a different approach that is independent of the planner used, focusing on modularity and placing special effort on the problem of extracting causal links from an existing plan.

Another system in which causal links are generated during the planning process is presented in ([19]). This work focuses on developing a temporal executive capable of modifying plans online by adding or removing causal links based on sensory information measuring resources and world states. Their approach uses a similar temporal plan representation, namely the STN ([5]). Our approaches differ in the manner of generating causal links, however. The work in ([19]) seems to obtain causal links as a consequence of the planning process, whereas we consider the problem of extracting causal links from plans independent of the planner used.

The approach taken in [2] develops the Quantitative Temporal Bayesian Network

(QTBN), a structure which can be used for execution monitoring of temporal plans in uncertain environments. Like the approaches presented above, [2] seems to obtain causal links from the planning process and is not concerned with the details of extracting them automatically from an existing plan.

The work in [27] is concerned with the monitoring of temporal plans encoded in a language called Golog. Should the execution monitor determine that the extension to the current execution trace is no longer feasible, the planner may backtrack and make different choices in the restartable plan. Our formulation of the execution monitoring problem differs greatly in that [27] does not seem to consider set-bounded duration temporal constraints, which is central to our set of algorithms.

2.3 Plan Executive Problem Statement

In this section, we provide an intuitive overview of the core technical problem solved in this thesis. Specifically, we will provide textual descriptions of a plan executive and an execution monitor, as well as plain-text and intuitive descriptions of the problems they solve. This section is intended to provide the broad, high-level overview of how different subcomponents fit together in this thesis. In the sections following this, we will define each of these subcomponents more formally and in order, thereby building up the requisites to present a formal definition of the plan executive and execution monitor in terms of these formalisms. These formal problem statements will be presented at the end of this section, once the requisites have been defined rigorously. We begin, however, by providing the more intuitive problem statements so that the formalisms that follow can be placed into context.

Intuitively, a plan executive is a system that is responsible for executing a plan in a robotic system. For plans with temporal constraints, plan execution entails two main sub problems. First, specific timing values for each action must be chosen and then the actions must be dispatched at those times. Second, the plan must be continually monitored to ensure that it is evolving properly. This task is of course execution monitoring.

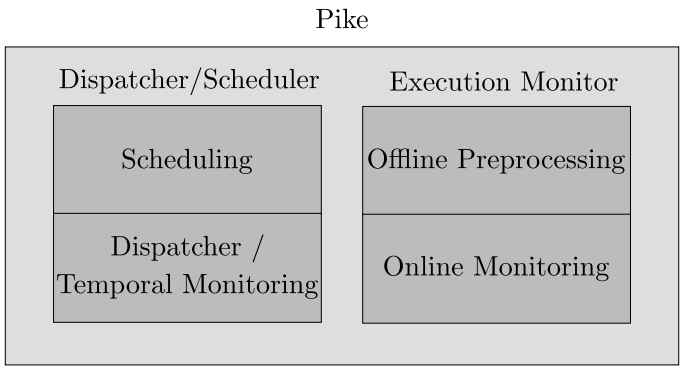

Figure 2-6: A high level overview of Pike, consisting of the dispatcher and execution monitor. The scheduler and dispatcher are loosely responsible for handing temporal constraints, and the execution monitor is responsible for handling state constraints.

Since plan execution and execution monitoring are so tightly intertwined and must both be considered when constructing a robust robotic system, we present the problem statement for a plan executive (and not just the execution monitor) in this section. In fact, we will regard the execution monitor as subroutine of the plan executive.

We thus introduce Pike, a plan executive capable of executing and monitoring temporal plans. Pike must be given as input a temporal plan, a description of actions in the plan, and the initial state and user specified goals. While executing the plan, Pike must also be given observations of the world in order to monitor the plan's progress. As its output, Pike must dispatch each of the activities in the plan at their proper times, and also monitor for progress and signal an error immediately if a disturbance is detected while executing the plan.

Pike is divided into two major subcomponents: 1.) a scheduler and dispatcher responsible for executing actions at their proper times, and 2.) the execution monitor that ensures that the state of the world is evolving as expected throughout the course of execution. Please see Figure 2-6 for an architecture diagram of Pike.

We will now elaborate on the inputs and outputs to Pike, starting with a description of a temporal plan. A temporal plan can be thought of as a sequence of actions to be carried out by a robotic system with temporal constraints that govern how long and when those actions may occur. It is key to note that these temporal plans are

flexible in terms of their timing constraints, as this temporal flexibility will play a key role in the formulation of our execution monitoring system. An example plan may contain constraints that can can be interpreted loosely as "2 to 3 minutes after the last action, move the red block on top of the blue block and take between 10 and 40 seconds to do so." A set of actions of this nature define a plan, and outline a way for the robot to achieve some goal given the current state of the world. We assume that we are given such a plan and do not consider the problem of generating plans in this thesis.

In addition to the temporal plan, Pike must also be given a description of the actions in the plan. This description must define requirements for each of the actions to run (preconditions), requirements that must hold true while the action is in progress (maintenance conditions), and changes to the world that occur as a direct result of executing the action (effects). As an example, let us consider the "pick up block from ground" for a robot. Preconditions for this include such requirements as the block being currently on the ground, the robot's gripper not holding anything else, and the block not having anything stacked on top of it that would preclude the robot's ability to pick up. All of these conditions must hold true in order for the "pick up block from ground" action to execute properly. A condition that must be maintained throughout the course of the action is that the robot must be able to reach the block at all times. If this is violated in the middle of the action, the action will fail. Finally, the "pick up block from ground" action makes several changes to the state of the world. After it finishes executing, the robot should be holding the block and the block should no longer be on the ground. The execution monitor must be given such operator descriptions of each action's preconditions, maintenance conditions, and effects in a generic way that can be applied to different blocks and objects in the world. Intuitively, the execution monitor needs this in order to back-infer potential problems that go wrong in the plan. For example, if some disturbance in the world makes it impossible for the block to ever be on the ground, the execution monitor must immediately infer that the "pick up block from ground" action will no longer be able to execute properly.

Pike must also be given the initial state of the world and the desired goal conditions to be satisfied from which the plan was generated. The initial state of the world is a complete set of facts that describe the state of the world - if the gripper is empty, which blocks are on the ground and which blocks are on other blocks, etc. The goal conditions need not specify the complete state of the world, but only provide constraints that the user desires to be true after execution (ex., "the red block is on top of the blue block"). The temporal plan provided to the execution monitor must achieve this goal starting from the initial conditions. That is, the sequence of actions specified by the plan must cause the state of the world to evolve in such a way that, by its end, the goal conditions are satisfied.

Finally, Pike must be given a continuous stream of state observations that describe the world at different points in time. This is crucial for the execution monitor to detect when disturbances occur.

Once the Pike plan executive is provided with all of these inputs, it proceeds to execute the plan by scheduling times for all of the activities in the plan (satisfying all temporal constraints), dispatching the activities at their proper times, and monitoring the execution for problems. Should it be determine during the midst of execution that the plan will no longer be able to succeed, Pike must immediately signal failure.

We can thus intuitively define the Pike problem statement as follows. Pike takes in the following inputs:

- A temporal plan containing actions to be executed,
- A set of operator descriptions describing the preconditions, maintenance conditions, and effects of the actions in the plan,
- The initial world state and the goal from which the plan was generated, and
- A continuous stream of world state observations which define the observed state in the present and past.

Pike generates the following outputs:

- Upon returning: a boolean value indicating whether the plan was executed successfully or if execution was terminated early because a problem occurred
- Upon returning: if the plan did not complete successfully, a conflict that illustrates the cause of failure
- During execution: Pike must dispatch all of the actions in the plan at the proper times so as to satisfy the temporal requirements of the plan and the preconditions/maintenance conditions of each action
- During execution: Pike must return either as soon as the plan finishes executing, or as soon as a relevant disturbance that threatens the plan is detected.

We formalize each of the above concepts in the following sections. Specifically, we discuss actions, temporal plans, the execution of temporal plans, and disturbances in the world in much greater detail. A specific problem statement rewritten in terms of formalisms for these concepts is provided afterwards. The intent of the this current section to place each of the following formalisms into context with respect to the problem we are solving, namely plan execution and monitoring.

2.3.1 Action Planning

In this section, we describe intuitively and formally the notion of actions in a plan. Actions are fundamental to execution monitoring and plan execution, as they provide the basis for conditions which must be monitored online. We begin by reviewing prior work in defining action models for robotic systems. We then select one of these models and formalize it in this section, providing the groundwork for future sections of this thesis.

STRIPS (Stanford Research Institute Problem Solver) planning was introduced in [11], and defines an action to be a name, a set of prerequisite facts that must be true for the action to run, as well as a set of facts that become true (the add list) and a set of facts that are no longer true (the delete list) after the action is run. Together, the add and delete lists specify changes to the world state that result from executing

the action. Given a set of initial facts about the world and a desired set of facts about the goal state, the STRIPS planning task is then to find an ordered sequence of actions such that the prerequisites of each action are true at the time the action is run, and that the actions modify the true facts starting from the given initial facts to take us to the goal state. Any such sequence of actions that is a solution to the STRIPs planning problem is called a complete plan [20].

Over the years, a number of additional planning domain languages have been developed. One of these, the Planning Domain Definition Language (or PDDL for short) is the one that we embrace in this thesis [13]. PDDL is expressed in a LISP-like syntax and adds a number of features to STRIPS, including typing, numerical fluents, time, and more. In this thesis, we focus on its semantics that are most similar in spirit to STRIPS with the addition of durative actions. From now on in this thesis, whenever we refer to PDDL, we refer to the subset of PDDL containing STRIPS and durative actions. Like STRIPS, PDDL allows actions to be specified in terms of their preconditions and effects. PDDL adds the notion that each action may have a range of possible durations. Additionally, we also consider maintenance conditions, or conditions that must hold true throughout the duration of the action. These preconditions, maintenance conditions, and effects are expressed in terms of *objects*, which are symbolic constants in PDDL that represent real-world objects. Examples of PDDL objects may be `RedBlock` or `Robot1`.

We now define the subset of PDDL that is relevant to this thesis. We begin with the PDDL predicate, which intuitively represents some property about the world.

Definition 2.3.1 (PDDL Predicate)**.** A *PDDL predicate* is denoted by

`(property` *`arg1 arg2 ...`*`)`

where `property` identifies the name of the PDDL predicate, and all of the subsequent parameters are known as arguments. The arguments may either all be *variables*, in which case the PDDL predicate is referred to as *ungrounded* or *uninstantiated*, or the arguments may all refer to specific *objects* in the world, in which case the PDDL

predicate is referred to as *grounded* or *instantiated*.

Examples of ungrounded PDDL predicates that are relevant to our block-stacking domain include `(on ?b1 ?b2)` and `(empty-gripper ?r)`. The arguments to these predicates are variables, which we denote with a question mark prefix. Two possible instantiations of these predicates are `(on RedBlock BlueBlock)` and `(empty-gripper Robot1)`, respectively. These PDDL predicates are grounded because their arguments are not variables; they are objects representing the real world. Collectively, the set of all grounded PDDL predicates represent the state of the world.

Definition 2.3.2. The *world state* at some time is the set of all grounded PDDL predicates in the world that hold true.

Now that we have defined properties of the world expressed in PDDL, we continue by describing how the world changes. PDDL actions modify the world state and are instantiated from PDDL operator schemas. We first begin by describing a PDDL Operator Schema, which intuitively represents an abstract action before it is instantiated with specific objects (i.e., akin to how "move" is different than "move this block"):

Definition 2.3.3 (PDDL Operator Schema)**.** A *PDDL operator schema* $\mathcal{O}$, also known as an *ungrounded PDDL operator* or an *uninstantiated PDDL operator*, is a tuple

$$\mathcal{O} = \langle name, \mathcal{X}, \textsc{Precond}(\mathcal{X}), \textsc{Maintenance}(\mathcal{X}), \textsc{Effects}(\mathcal{X}) \rangle$$

where *name* is a string representing the name for this operator, $\mathcal{X}$ is a set of variables (later, each variable will be bound to a specific object in the world to ground this operator), $\textsc{Precond}(\mathcal{X})$ is a conjunction of ungrounded predicates over the variables in $\mathcal{X}$ representing the preconditions of the operator, $\textsc{Maintenance}(\mathcal{X})$ is a conjunction of ungrounded predicates over the variables in $\mathcal{X}$ representing conditions that must hold true while the operator is being executed, and $\textsc{Effects}(\mathcal{X})$ is a conjunction of literals of ungrounded predicates over the variables in $\mathcal{X}$ representing the modifications to the world that result from executing this operator. Note that since

```
(:durative-action pick-up-block-from-ground
  :parameters (?r - robot ?t - block)

  :duration  (and (>= ?duration 10) (<= ?duration 30))

  :condition (and (at start (clear-above ?t))
                  (at start (empty-gripper ?r))
                  (over all (can-reach ?r ?t))
                  (at start (on-ground ?t)))

  :effect    (and (at end (not (empty-gripper ?r)))
                  (at end (not (on-ground ?t)))
                  (at end (holding ?r ?t))
                  (at end (not (clear-above ?t))))
```

Figure 2-7: An example of a PDDL version 2.1 operator schema representing the "pick-up-block-from-ground" action. Note precondition and effect lists, which when bound with values specified in the parameters list, yield a grounded or instantiated PDDL operator. PDDL uses a LISP-like syntax.

conjunctions of literals (i.e., conjunctions of a or $\neg a$ where a is a PDDL predicate) are used, $\textsc{Effects}(\mathcal{X})$ describes both the add and delete lists from STRIPS.

An example of a PDDL operator schema is shown in Figure 2-7 for the generic action of picking up a block from the ground. The parameters in a PDDL operator schema are abstract variables that can be *bound* to specific objects in the world. A PDDL operator schema conceptually defines a set of possible actions that can be produced, and a binding specifies to which specific objects in the world the action applies. Since many different bindings are possible, an operator schema represents many different possible specific actions. We can produce a specific *PDDL action* by substituting the binding assignments into an operator schema's precondition and effects predicates. This is a process known as *grounding* or *instantiation*, and yields a PDDL action as a result.

We formally define a *binding* and *PDDL action* as follows:

Definition 2.3.4 (Binding)**.** The *binding* of an ungrounded PDDL operator $\mathcal{O}$ refers to making a full assignment to each of the variables in the parameters $\mathcal{X}$ of $\mathcal{O}$.

Formally, a binding is a set of pairs $\{(x_i, o_i), ...\}$ for each $x_i \in \mathcal{X}$ representing $x_i = o_i$, where o_i is an object in the world.

Definition 2.3.5 (PDDL Action)**.** A *PDDL action* $\boldsymbol{\alpha}$, other known as a *grounded PDDL operator* or an *instantiated PDDL operator*, is a tuple

$$\boldsymbol{\alpha} = \langle name, \textsc{Preconds}, \textsc{Maintenance}, \textsc{Effects} \rangle$$

where *name* is the name of the PDDL action (same as the operator from which it is derived), PRECONDS is a set of grounded PDDL predicates representing a conjunction of preconditions for the action, MAINTENANCE is a set of grounded PDDL predicates representing a conjunction of maintenance conditions that must hold true throughout the duration of the operator, and EFFECTS is a conjunction of literals of grounded PDDL predicates representing the effects of each action. When we refer to the *conditions* of a PDDL action, we mean all of its preconditions and maintenance conditions (namely PRECONDS $\cup$ MAINTENANCE).

Please note that, unlike an ungrounded PDDL operator, a PDDL action's preconditions and effects are no longer dependent on the operator's parameters. Rather, they are grounded to specific objects in the world. For example, taking again the general PDDL operator schema in Figure 2-7 and the binding $\{(\texttt{?r}, \textit{Robot1}), (\texttt{?t}, \textit{RedBlock})\}$, we produce a specific PDDL action, referred to in LISP-like syntax, `(pick-up-block-from-ground` *Robot1 RedBlock*`)`. This PDDL action has the preconditions

```
(and (clear-above RedBlock)
     (empty-gripper Robot1)
     (can-reach Robot1 RedBlock)
     (on-ground RedBlock))
```

and the effects

```
(and (not (empty-gripper Robot1))
     (not (on-ground RedBlock))
```

```
(holding Robot1 RedBlock1)
(not (clear-above RedBlock1))))
```

The notion of PDDL operator schemas and actions are central to this thesis. Recall that we mentioned earlier that Pike must take as input a description of each action in the plan. These descriptions are PDDL operator schemas.

2.3.2 Temporal Plans

Now that we have defined actions formally, we proceed to piece them together (along with timing information) to form plans. Pike takes these plans as input, and is responsible for executing and monitoring them throughout the course of execution.

Many classical planners consider a plan to be an ordered sequence of actions which, when carried out starting at some fixed world state, will yield a given desired state. For example, a totally or partially ordered sequence of PDDL actions would be considered a plan. In this section, we discuss an augmentation to these classical plans that includes an explicit notion of time and flexibility. Intuitively, we associate a flexible start time and duration with each action in the plan.

The techniques in this section are based on previous work in formulating temporal constraint networks. The general Temporal Constraint Network (TCN) is formulated in [5], and is additionally simplified into the Simple Temporal Network (STN). These approaches associate set-bounded binary constraints between nodes in a graph called events, each of which represents a specific point in time.

Intuitively, a *temporal plan* is a set of actions to which temporal constraints have been applied. These timing requirements are imposed over *events* in the plan, which represent time points such as the start or finish of executable actions (see Figure 2-8). We use *simple temporal constraints* to represent our time constraints for their intuitive nature as well as for their balance between expressiveness and computational tractability ([5]). A simple temporal constraint specifies a lower bound and upper bound duration on the time difference between two events. For example, we could specify that the beginning of a "start cooking dinner" event must occur between

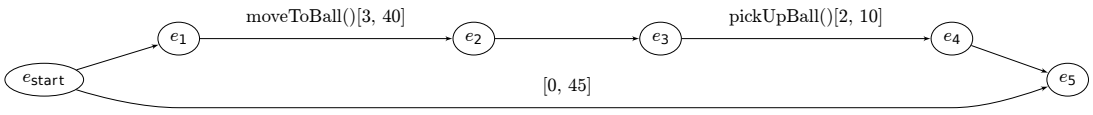

Figure 2-8: A temporal plan can be interpreted visually as a graph. The circular vertices represents events $e \in \mathcal{E}$. Directional arcs connecting vertices contain instantiated actions as well as temporal constraints in the form $[lb, ub]$. Arcs without labels have no actions, and default to a temporal constraint of $[0, 0]$ (the two events occur at exactly the same time).

60 and 120 seconds after a "finish finding ingredients" event. We define the *simple temporal constraint* as follows:

Definition 2.3.6 (Simple Temporal Constraint)**.** A *simple temporal constraint* represents a set-bounded duration between a pair of events. Formally, a simple temporal constraint $c \in \mathcal{C}$ is a tuple $\langle e_s, e_f, lb, ub \rangle$ where $e_s, e_f \in \mathcal{E}$, and lb (lowerbound) and ub (upperbound) are values in $\mathbb{R}$ such that $ub \geq lb$. The meaning of the constraint is that $lb \leq t_{e_f} - t_{e_s} \leq ub$.

We are now equipped to define the *temporal plan*:

Definition 2.3.7 (Temporal Plan)**.** A *temporal plan* $\mathcal{P}$ is a tuple $\langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$, such that:

- $\mathcal{E}$ is a set of *events*. Each event $e \in \mathcal{E}$ is associated with a specific point in time $t_e \in \mathbb{R}$. Additionally, there is a distinguished event e_{start} that represents the first event and for which $t_{e_{start}} = 0$.
- $\mathcal{C}$ is a set of *simple temporal constraints* over the time points in $\mathcal{E}$.
- $\mathcal{A}$ is a set of *actions* in the plan.

The definition of a temporal plan refers to *actions*, which we define below. Intuitively, an action specifies a PDDL action (i.e., an instantiated predicate) as well as a start event and end event from the plan that constrain the duration of the action.

Definition 2.3.8 (Action)**.** Given a plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$, an *action* $a \in \mathcal{A}$ is a tuple $\langle \boldsymbol{\alpha}, e_s, e_f \rangle$ where $\boldsymbol{\alpha}$ is a PDDL action (i.e., an instantiated PDDL operator) and $e_s, e_f \in \mathcal{E}$. The *start time* of a is t_{e_s}, or the time of event e_s, and the *finish time* of a is t_{e_f}. The duration of a is therefore t_{e_f} - t_{e_s} and, we constraint this to be strictly positive. In other words, $t_{e_f} - t_{e_s} > 0$.

It is critical to note that these plans are least-commitment in the temporal sense. By allowing ranges in the durations between events, a dynamic dispatcher is capable of deferring commitment to a specific schedule until the latest point, thereby maximizing its flexibility.

2.3.3 Execution of Temporal Plans

Now that we have introduced temporal plans and PDDL actions, we proceed to describe the execution of these plans. We begin by defining the notion of an execution trace and a schedule for a temporal plan. We then proceed to describe a method for evaluating the preconditions and maintenance conditions of actions in a plan with respect to the worlds state by using what we call *world state functions*. The culmination of this section follows, with definitions of what it means for a schedule to be *executable* and an execution trace to be *healthy*. These two key definitions play an important role in the execution monitor's output.

Execution Traces and Schedules

We begin by defining the execution trace for a temporal plan. Since our monitoring runs continuously online, at most points during execution the plan will not be finished executing. Specifically, of the events in $e \in \mathcal{E}$ for a temporal plan, only a subset of these events will have been dispatched thus far by time t_{now}. The remaining events will occur in the future, scheduled at times for which $t \geq t_{now}$. Intuitively, the notion of an execution trace T_{trace} captures the subset of events that have been dispatched up to t_{now}, and orders these events by their dispatch times:

Definition 2.3.9 (Execution Trace of a Temporal Plan)**.** An *execution trace* T_{trace}

for a temporal plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ is a partial assignment to the event times in $\mathcal{E}$. Formally, an execution trace T_{trace} up to time t_{now} is a set of pairs $\{(t_1, e_1), (t_2, e_2), ...\}$ such that t_i is the execution time of e_i (denoted $t_{e_i} = t_i$), $t_i \leq t_{now}$, and $\bigcup_i e_i \subseteq \mathcal{E}$. Without loss of generality we order these assignments such that $t_i \leq t_{i+1}$.

Once execution has completed, all of the events in $\mathcal{E}$ will have been dispatched and assigned specific time values. As a result, the execution trace at the end of execution must contain an assignment for every event. We call this full-assignment a schedule T_{full}:

Definition 2.3.10 (Schedule for a Temporal Plan)**.** A *schedule* T_{full} for a temporal plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ is a full assignment to the event times in $\mathcal{E}$. Formally, a schedule T_{full} is an execution trace $\{(t_1, e_1), (t_2, e_2), (t_3, e_3), ...\}$ such that $\bigcup_i e_i = \mathcal{E}$, and all e_i are unique.

A schedule is called *consistent* if it satisfies all of the temporal constraints of the plan. For Pike to be successful, the schedule it produces must be consistent.

Definition 2.3.11 (Consistent Schedule)**.** A schedule for a plan is called *consistent* or *temporally consistent* if it satisfies all of the temporal constraints of the plan. Formally, a schedule $T_{full} = \{(t_{e_i}, e_i), ...\}$ for a temporal plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ is consistent if the event time assignments satisfy all $c \in \mathcal{C}$.

Finally, we define the notion of an *extension* to an execution trace. Intuitively, an extension to an execution trace is a full schedule in which all future events left unassigned in the execution trace are now assigned values. Given an execution trace for time up to t_{now}, a full schedule extension can be constructed by assigning values $t > t_{now}$ to the unassigned events in the execution trace.

Definition 2.3.12 (Extension to an Execution Trace)**.** An *extension to an execution trace* T_{trace} is a schedule (i.e., a full assignment to all event times) such that those events not assigned in T_{trace} are assigned values. Formally, an extension T_{full} to an execution trace T_{trace} is a schedule such that every assignment $(t_i, e_i) \in T_{trace}$ is also in T_{full}.

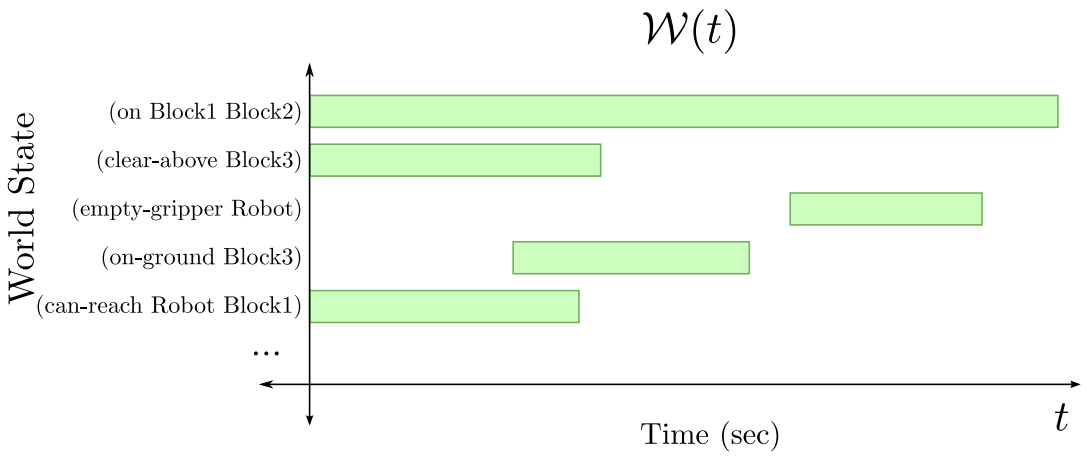

Figure 2-9: A visualization of an example world state function $\mathcal{W}(t)$. Time progresses along the x-axis, and the y-axis represents world state. Each of the labels on the y-axis represents a particular PDDL predicate that can be either TRUE or FALSE. The colored bars indicate the duration for which the given predicate is TRUE. Note that by taking a vertical slice representing some fixed time, we have the world state at that time.

World State Functions

Thus far, we have mainly been concerned with temporal constraints. A consistent schedule satisfies the temporal constraints of a plan. We would like to define a similar notion for a schedule, called an *executable* schedule, that takes into account the pre-conditions and maintenance conditions of actions in the plan. Intuitively, we will call a schedule executable if it is both temporally consistent and the preconditions and maintenance conditions of actions will be met when when each action is dispatched.

We approach the task of rigorously defining executability using what we call *world state functions*. A *world state function* defines the state of the world as evaluated at different points in time. Recall that the world state is the complete set of all PDDL predicates that hold TRUE. For any time t, a *world state function* returns a world state for that time (see Figure 2-9).

Definition 2.3.13 (World State Function)**.** A *world state function* $\mathcal{W}(t)$ returns a world state (i.e., thea set of all PDDL predicates that are TRUE) as a function of time. If some predicate p is TRUE at some time t according to $\mathcal{W}$, we write that

$p \in \mathcal{W}(t)$. Otherwise, if p is not in the set of PDDL predicates at time t and hence does not hold TRUE, we write that $\neg p \in \mathcal{W}(t)$.

There are several different world state functions that each have different properties and make different assumptions about the world: $\mathcal{W}_{obs}(t)$, $\mathcal{W}_{ideal}(t)$, and $\mathcal{W}_{pred}(t)$. We introduce these below, and make use of them in our definitions and proofs throughout this thesis. Of these three, $\mathcal{W}_{pred}(t)$ is the most important, and we define $\mathcal{W}_{obs}(t)$ and $\mathcal{W}_{ideal}(t)$ mainly in order to define $\mathcal{W}_{pred}(t)$.

First, and perhaps the most conceptually simple, is the observed world state function denoted $\mathcal{W}_{obs}(t)$. This world state function is only valid for the past up to the present ($t \leq t_{now}$), and returns state observations that were recorded in the past. It essentially "plays back" state that was observed previously. These observations may have come from a hybrid estimation module, or any other means of estimating state.

Definition 2.3.14 (Observed World State Function)**.** Given the current time t_{now}, the observed world state function $\mathcal{W}_{obs}(t)$ returns the world state that was observed in the past. The domain of $\mathcal{W}_{obs}(t)$ is $t \leq t_{now}$.

Next, in contrast to the observed world state function that is based completely on observed state, we introduce the ideal world state function, denoted $\mathcal{W}_{ideal}(t)$. The ideal world state function, when given an initial world state $\mathcal{W}_0$, a set of actions $\mathcal{A}$, and a schedule T_{full}, returns the state over time that would "ideally" result assuming that each action $a \in \mathcal{A}$ produces its effects precisely at its finish time as defined in T_{full}. No disturbances, stochasticity, or other forms of non-determinism are taken into account by $\mathcal{W}_{ideal}(t)$. It is assumed that each predicate holds its value indefinitely, unless changed by the effect of some action (see Figure 2-10). We define $\mathcal{W}_{ideal}(t)$ formally as follows:

Definition 2.3.15 (Ideal World State Function)**.** Given a plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$, a corresponding schedule $T_{full} = \{(t_1, e_1), (t_2, e_2), (t_3, e_3), ...(t_N, e_N)\}$ ordered such that $t_i \leq$

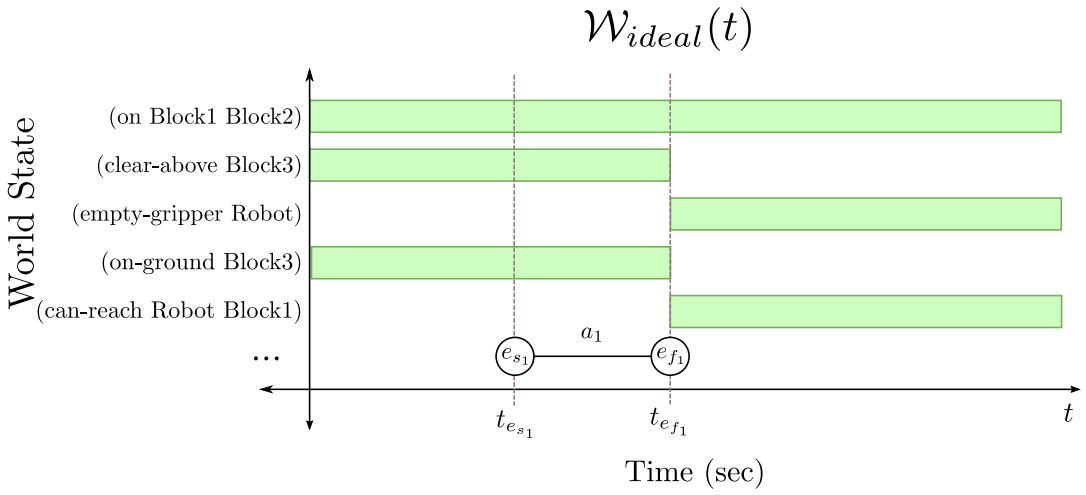

Figure 2-10: A visualization of $\mathcal{W}_{ideal}(t)$ superimposed over an action a_1. In $\mathcal{W}_{ideal}$, predicates hold their values indefinitely unless changed by the effect of some action. This instantaneous change occurs precisely at the action's finish event. This is illustrated in the image above. Note that at time $t_{e_{f_1}}$, when a_1's finish event is scheduled, some predicates change truthfulness. They retain their values for all other times when there is no action finishing.

t_{i+1}, and an initial world state $\mathcal{W}_0$, we first define a discrete sequence $(\mathcal{W}_0, \mathcal{W}_1, ..., \mathcal{W}_N)$:

$$\mathcal{W}_{i+1} = \begin{cases} \textsc{ApplyEffects}(\mathcal{W}_i, \boldsymbol{\alpha}) & \text{if } e_{i+1} = e_f \text{ for some } a = \langle \boldsymbol{\alpha}, e_s, e_f \rangle \in \mathcal{A} \\ \mathcal{W}_i & \text{otherwise} \end{cases}$$

This sequence starts from the given initial world state $\mathcal{W}_0$. The ApplyEffects function above adds all of the add-effects of the PDDL action $\boldsymbol{\alpha}$ to the given world state while removing all of the delete-effects of $\boldsymbol{\alpha}$. Given this discrete sequence of states, we now map them to piecewise time intervals as follows to produce $\mathcal{W}_{ideal}(t; ...)$:

$$\mathcal{W}_{ideal}(t; \mathcal{W}_0, \mathcal{A}, T_{full}) = \begin{cases} \mathcal{W}_0 & \text{if } t < t_1 \\ \mathcal{W}_i & \text{if } t \in [t_i, t_{i+1}) \\ \mathcal{W}_N & \text{if } t \geq t_N) \end{cases}$$

The ideal world state function is useful for predicting the future, making nominal

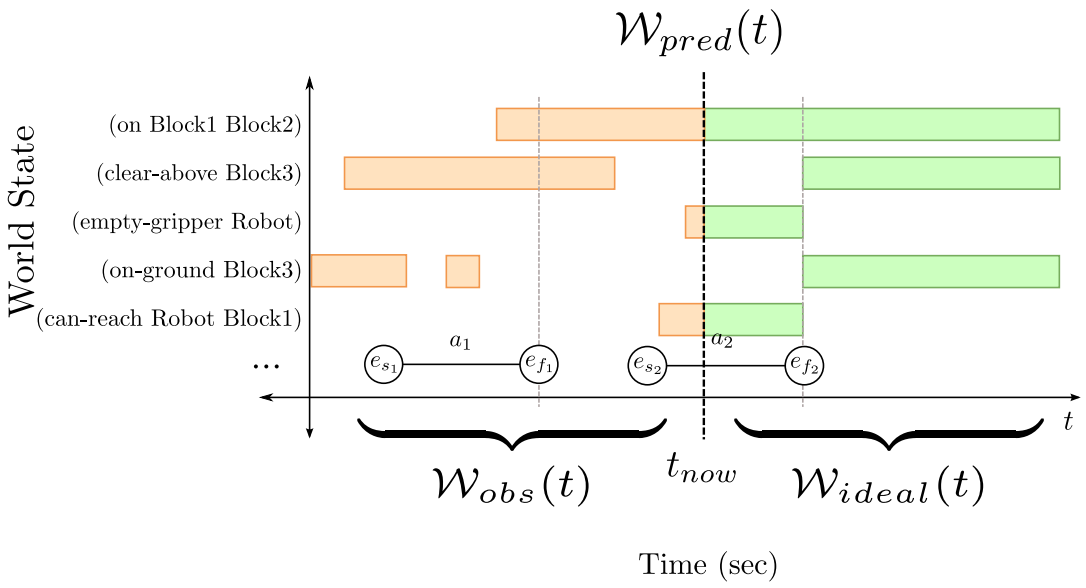

Figure 2-11: An illustration of the $\mathcal{W}_{pred}(t)$ world state function, which combines past observations with future predictions assuming an ideal world. For all $t \leq t_{now}$, $\mathcal{W}_{pred}(t) = \mathcal{W}_{obs}(t)$, namely the observed state of the world (as could be inferred by a hybrid estimation module). This is illustrated above in orange. Note that here the world state may not behave cleanly, as it reflects actual world state (and the real world is certainly not ideal). For all $t > t_{now}$, $\mathcal{W}_{pred}(t) = \mathcal{W}_{ideal}(t)$ starting from the observed state $\mathcal{W}_{obs}(t_{now})$ at time t_{now}. Thus $\mathcal{W}_{pred}$ is "continuous" about $t = t_{now}$; any predicate observed to be TRUE at t_{now} (any orange bar intersecting t_{now}) will be assumed to continue holding TRUE until some action modifies it (depicted at the finish event of a_2). Note that by combining past observations and future predictions, the domain of $\mathcal{W}_{pred}(t)$ is $-\infty < t < \infty$.

assumptions about the behavior of actions in a plan. The observation world state function is useful for returning what the state actually was in the past. We now combine these two functions in order to produce the most useful world state function for execution monitoring. Intuitively, we would like a world state function that, for the past, returns the observations, and for the future, returns predictions extrapolated forward from the observed current state of the world. We call this "predicted" world state function $\mathcal{W}_{pred}(t)$ since it predicts the state of the world currently for all time past and present. We construct $\mathcal{W}_{pred}(t)$ by concatenating $\mathcal{W}_{obs}(t)$ and an instance of $\mathcal{W}_{ideal}(t)$ starting from the current state splitting between them at the present t_{now}:

Definition 2.3.16 (Predicted World State Function)**.** Given a world state function for past observations $\mathcal{W}_{obs}(t)$, the current time t_{now}, a set of actions $\mathcal{A}$, and a schedule T_{full}, we define the predicted world state function $\mathcal{W}_{pred}(t)$ as follows:

$$\mathcal{W}_{pred}(t; \mathcal{W}_{obs}, t_{now}, \mathcal{A}, T_{full}) = \begin{cases} \mathcal{W}_{obs}(t) & \text{if } t \leq t_{now} \\ \mathcal{W}_{ideal}(t; \mathcal{W}_{obs}(t_{now}), \mathcal{A}, T_{full}) & \text{if } t > t_{now} \end{cases}$$

An intuitive illustration of $\mathcal{W}_{pred}(t)$ is shown in Figure 2-11. Note that for all time in the past, $\mathcal{W}_{pred}(t)$ replays the past observations. For all time in the future $\mathcal{W}_{pred}(t)$ assumes the world behaves ideally and takes the current world state as measured by $\mathcal{W}_{obs}(t_{now})$ as the initial state. For each value of t_{now}, $\mathcal{W}_{ideal}$ is "seeded" with an initial value in the above definition so that its future predictions during $t > t_{now}$ will be based on the current state at t_{now}

Executable Schedule and Healthy Execution Trace

Now that we have set up the necessary formalisms for evaluating the truthfulness of PDDL predicates at various points in time (namely, the world state functions defined above), we proceed to define what it means for a schedule to be executable.

Intuitively, a schedule is *executable* if it is temporally consistent, and if each action's preconditions and maintenance conditions will be met at the appropriate times as defined by the schedule. However, since there are different ways of measuring the state of the world (observed, predicted, etc.) we define the notion of a schedule being *executable under* $\mathcal{W}$.

Definition 2.3.17 (Executable Schedule)**.** A schedule T_{full} for a plan is called *executable under* $\mathcal{W}$ if it is temporally consistent, and if all of the preconditions and maintenance conditions of the actions in $\mathcal{P}$ hold true at their proper times according to $\mathcal{W}$ and T_{full}. In other words, a schedule T_{full} for a plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ is executable under $\mathcal{W}$ if T_{full} satisfies all of the temporal constraints in $\mathcal{C}$, and for each action $a = \langle \boldsymbol{\alpha}, e_s, e_f \rangle \in \mathcal{A}$ the following two conditions hold: 1.) $\textsc{Precond}(\boldsymbol{\alpha}) \subseteq \mathcal{W}(t_{e_s})$, and 2.) $\textsc{Maintenance}(\boldsymbol{\alpha}) \subseteq \mathcal{W}(t)$ for $t_{e_s} < t < t_{e_f}$.

The notion of an executable schedule checks to see if each action's preconditions and maintenance conditions (the requirements for that operator to run) hold according to the given world state function. Depending on the world state function used for this evaluation, the idealistic effects of actions or past recorded observations will provide the basis from which these judgements are made. For example, if a schedule is executable under $\mathcal{W}_{ideal}$, this means that we take into account the effects of all the actions at the proper times in evaluating the other actions' conditions.

The goal of the plan executive is to dispatch events such that the schedule resulting at the end of execution T_{full} is executable under $\mathcal{W}_{obs}$. Namely, all temporal and precondition/maintenance condition constraints were satisfied during execution. Similarly, the goal of the execution monitor is to judge whether or not the execution trace at the current time instant t_{now} can possibly be extended into a full schedule that is executable. This leads us to the notion of a *healthy* execution trace.

Definition 2.3.18 (Healthy Execution Trace)**.** An execution trace of a plan $\mathcal{P}$ up to t_{now} is *healthy* if and only if there exists a full-schedule extension to the trace that is executable under $\mathcal{W}_{pred}$.

If the execution trace up to some t_{now} is healthy, we can expect that the plan executive will be able to successfully complete the plan and satisfy all temporal constraints and precondition/maintenance condition requirements - both those for actions occuring in the past, and also for those in the future. However, it is important to note several important assumptions we make when we talk about an execution trace's healthiness. We assume that all actions in the future produce their effects as promised at their finish times, and that predicates hold their values until unchanged. However, this means that an execution trace's health status must be taken with a grain of salt. If an execution trace is deemed to be healthy, we mean that it will proceed successfully if the above assumptions are true. We expect an execution trace that is healthy to finish properly in the absence of any data to the contrary. However, as our models afford no way to predict disturbances in the future, it is very possible that a healthy execution trace may later become unhealthy should a disturbance occur in the fu-

ture. A trace initially measured as healthy may actually fail as a result of unmodeled disturbances. Analogously, an execution trace that is deemed to be unhealthy may actually turn out to be executable, should a second disturbance occur that corrects the original (i.e., "two wrongs may make a right"). With no way of obtaining an oracle for the future, we accept these limitations in this thesis.

2.3.4 Disturbances in the World

Thus far in introducing action planning and temporal plans, we have imagined a perfect world void of disturbances. However, since the real world often changes in unpredictable ways, we introduce a model for disturbances and unexpected errors. In this section, we build upon the notion of the world state function presented earlier. We first introduce a disturbance, and then provide intuition differentiating the notion of a relevant disturbance from that of an irrelevant disturbance (these will be formalized in later sections).

First, we begin with the definition of a disturbance.

Definition 2.3.19 (Disturbance)**.** A *disturbance* with respect to a plan is a change to the world state that is not described by the actions of the plan. Formally, given an initial state $\mathcal{W}_0$ and a plan $\mathcal{P}$, a disturbance occurs at time t if t is the first time at which $\mathcal{W}_{obs}(t;...) \neq \mathcal{W}_{pred}(t;...)$. In other words, any instant at which the world state suddenly deviates with respect to what would be expected via $\mathcal{W}_{pred}$ is called a disturbance.

There are many kinds of disturbances that satisfy this intentionally broad definition. PDDL actions that fail to yield their promised effects (ex., the pick-up-block action failing to actually pick up the block) is an example of a disturbance. However, disturbances need not only be caused by action failures - they can also be caused by unexpected and spontaneous events. For example, a toddler grabbing the block from out of the robot's gripper is also considered a disturbance. Broadly speaking, any change that would not be predicted naturally from the plan is a disturbance.

We further distinguish disturbances into types: relevant and irrelevant. We design

our execution monitoring to only detect the relevant disturbances and to ignore the irrelevant ones. Intuitively, a relevant disturbance is a change that will affect the future outcome of the plan. An irrelevant disturbance, while indeed changing the world state, will not require replanning. An example of an irrelevant disturbance would be removing some never-used blocks from the work area. Since they are never used, these blocks will not interfere with the robot's plan.

In the ideal world where there are no relevant disturbances, replanning will never be required and an execution monitor is unnecessary. The sole agent of change assumption used by classical planners would hold perfectly.

2.3.5 Conflicts

The final formalism we introduce is the notion of a conflict. A conflict can be thought of as set of constraints that are logically inconsistent. If Pike encounters an error during execution, it returns a conflict that is indicative of the problem.

Conflicts have been used in many different areas of artificial intelligence. For example, conflicts can be used in the process of isolating faults in a system ([4]) or to improve search efficiency by dramatically pruning the search space ([31]). As will be discussed in the following chapter of this thesis in greater detail, Pike returns a conflict so that a planning algorithm calling Pike as a subroutine will have the ability to generate new plans efficiently.

Formally, we define a conflict as follows:

Definition 2.3.20 (Conflict)**.** A *conflict* is a set of constraints or observations that entail a logical inconsistency. Put another way, it is not logically possible for all constraints in the conflict to hold TRUE simultaneously.

For example, consider the case of a block stacking robot in the midst of building a tower. Further suppose that the robot accidentally drops a block that it is trying to stack on the tower. Then, the set of constraints {"Action 1 puts the block on the tower", "the block will remain on the tower", "Action 1 has executed", "Observation: the block is not on the tower", "the sky is blue"} is a conflict, because all of

these constraints cannot be simultaneously TRUE. We notice however that the final constraint is superfluous. This leads us to define the minimal conflict:

Definition 2.3.21 (Minimal Conflict)**.** A *minimal conflict* is a conflict such that the removal of any constraint from the set would resolve the conflict.

The above conflict is not minimal, since removing "they sky is blue" still results in a conflict. However, if we do remove it to yield {"Action 1 puts the block on the tower", "the block will remain on the tower", "Action 1 has executed", "Observation: the block is not on the tower", "the sky is blue"}, then this is indeed a minimal conflict. Should any of these constraints be removed, it would be possible for all constraints in the set to then hold TRUE simultaneously. Intuitively, a minimal conflict therefore captures the essence of the inconsistency by only admitting relevant constraints.

As will be discussed in our algorithmic sections in greater detail, Pike is capable of returning conflicts containing state and temporal constraints from the plan.

2.3.6 Formal Pike Problem Statement

At this point, we have described an intuitive problem statement for the Pike plan executive which uses the execution monitor as a core subcomponent. We have also discussed the necessary formalisms to describe the inputs and outputs of Pike rigorously. This section is devoted to tying all of these pieces together; we now express the intuitive problem statement of Pike in formal terms now that all of the necessary requisites have been defined.

Pike takes in the following inputs:

- A temporal plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ that is to be executed
- A set $\{\mathcal{O}_1, \mathcal{O}_2, ...\}$ of PDDL operator schema that describe the preconditions, maintenance conditions, and effects of the actions in $\mathcal{P}$
- The initial world state $\mathcal{W}_0$

- A conjunction of PDDL predicates representing the goal g from which the plan $\mathcal{P}$ was generated
- A stream of world state observations $\mathcal{W}_{obs}(t)$ valid for all $t \leq t_{now}$ which define the observed state in the present and past.

Pike obeys the following while in the midst of executing $\mathcal{P}$:

- Pike must dispatch events $e \in \mathcal{E}$ at their proper times so as to satisfy all of the temporal constraints in $\mathcal{C}$, and
- Pike must return immediately if the execution trace becomes unhealthy. Formally, should the execution trace transition from being healthy to unhealthy at time t, then Pike must return at time t.

Additionally, Pike generates the following outputs when it returns:

- A boolean value SUCCESS indicating whether the execution completed successfully. SUCCESS = TRUE if and only if all events $e \in \mathcal{E}$ were dispatched and the resulting schedule is executable under $\mathcal{W}_{obs}$
- A set of minimal conflicts Γ, only if SUCCESS = FALSE. Γ may contain conflicts with temporal constraints as well as and precondition/maintenance condition constraints from $\mathcal{P}$.

2.4 Causal Link Execution Monitoring System

Now that we have defined the problem statements for the Pike plan executive, we present our approach to solving this problem. From a high level, our approach consists of two sets of algorithms: a set of offline algorithms that run before the plan is dispatched by the plan executive, and a second set of online algorithms that run while the plan is executing. We provide intuition regarding how both sets of algorithms work, discussing both execution monitoring and plan execution.

The offline stage of the execution monitor infers the intent or rationale behind actions in a plan. Intuitively, we extract such information as "Action 1 is what

produces a predicate required by some later Action 2." This idea is called a *causal link* and allows us to infer that Action 1 is needed because it is the provider of the predicate needed by Action 2 ([20]). Most importantly, if Action 1 were to fail, or if the predicate were to suddenly disappear during some time between Action 1 and Action 2, we know that Action 2 would fail because its preconditions would not be met. By extracting causal links from a plan, we can infer the requirements of each action and also compute a set of conditions that must hold true under nominal circumstances for a plan to be executable. The goal of the offline preprocessing stage is to deduce these causal links, whose predicates will then be continuously monitored during the online phase.

Prior work assumes that a unique set of causal links can be determined from the planning process, or can be computed uniquely from a totally-ordered plan ([24], [21]). It is these conditions that are then monitored during the online phase. However, this story is complicated by the fact that for an arbitrary temporal plan, a unique set of causal links cannot always be extracted. Rather, uncertainty in when actions complete can lead to uncertainty in which actions will actually produce the predicates required by later consuming actions. In other words, it may not be possible to guarantee which action must be the producer for a given consumer action during the offline stage when a precise schedule is not known. This idea of a non-unique set of causal links is a novel development in this thesis. Our approach to solving this problem is to extract sets of *candidate causal links* during the offline preprocessing phase, which will be refined during the online phase as execution progresses and more information is known.

During the online stage, the plan executive dispatches actions in the plan in such a way as to satisfy the plan's temporal constraints. The executive consists of two components: a *scheduler* that selects appropriate times for each event in the plan, and a *dispatcher* which is responsible for executing those events at the scheduled time. During this process, the execution monitor continually checks a set of *active monitor conditions* representing a minimal set of conditions that must hold true in the world in order for the current execution trace to be healthy. As mentioned above,

the offline preprocessing stage extracts sets of candidate causal links. During online execution, one candidate from each of these sets is *activated* at the appropriate time and monitored during a certain period. Intuitively, the execution monitor activates the causal link with the latest producer finish event from within each candidate set. This strategy exploits scheduling information from the execution trace in order to guarantee that the other causal links in the candidate set must have had earlier producer finish events, and would hence not have been the sole cause of producing the consumer's precondition.

In the following sections, we formally present the notion of causal links, as they are the cornerstone of our execution monitoring approach. We then describe the process through which we extract causal links from least-commitment temporal plans. We proceed by describing the online monitoring algorithms. Each section will be accompanied with relevant proofs of various properties, such soundness and completeness.

Finally, we conclude with a theoretical performance analysis of these algorithms, as well results from experimental validation on a number of different domains.

2.4.1 Algorithmic Approach

In this section, we give a more detailed overview of the specific algorithms and approaches we propose for our plan executive, Pike, and our execution monitor. The previous sections discussed some goals for some of these algorithms. We now delve further, illustrating from a high level our approach to designing the algorithms that will meet these goals. Each of the subsequent sections will provide details, formalisms, and proofs of correctness for these subpieces. Our aim is to illustrate how our algorithms function from a birds-eye view in this section, in order to make the context of each subpiece more clear in the later sections.

Let us begin by recalling that the offline portion of the execution monitor is responsible for generating candidate sets of causal links, each of which represents the idea that some producing action a_1 produces some predicate p that is required by some later consuming action a_2, and that no other action a_{threat} occurs in time between a_1 and a_2 that asserts $\neg p$. Once sets of causal links are extracted, they

will be monitored using an online algorithm that is closely intertwined with the plan executive's online dispatching algorithm.

The plans we wish to operate on and extract causal links from are temporally flexible. Due to the use of flexible simple temporal constraints as opposed to rigid time requirements (i.e., "action a_1 may start between 1 and 10 minutes in the future, and take between 5 and 20 seconds" versus "action a_1 must start in exactly 2 minutes and take exactly 3 seconds"), we cannot in general determine a total ordering of each action in the plan. Before the plan is dispatched, we do not yet know at what time each action will be executed and what its duration will be. This presents a challenge for causal link extraction, which requires that the producing action temporally precede the consuming action. However, using a set of algorithms developed in prior work that are designed to reason over networks of simple temporal constraints, we are able to make some guarantees in some situations about some event orderings ([5]). We make heavy use of this in our candidate causal link extraction algorithms.

Given a temporal plan $\mathcal{P}$, it is possible to expose explicit constraint by converting the $\mathcal{P}$ to a *distance graph* and subsequently running an all-pairs shortest path (APSP) algorithm such as Floyd Warshall on this distance graph ([5]). We will not describe the details of this well-known process in this thesis, but will use it as a subroutine in our algorithms. Once computed, the distance graph supports a number of useful queries, such as asking if one event is guaranteed to proceed some other event in all possible temporally consistent schedules of the plan.

As mentioned above, we cannot in general determine a temporal total ordering over all actions in the plan to determine which actions will come first. We can, however, use the APSP query functions to construct a partial ordering over all actions. Our causal link extraction algorithm begins by processing $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ in order to produce what we refer to as an *action graph*, which is a graph where each vertex represents an action $a \in \mathcal{A}$, and each directed edge represents an ordering constraint. Note that some pairs of vertices in this graph may not be connected with an edge if the APSP is inconclusive at providing an ordering for those actions.

Using this action graph, we are able to extract candidate sets of causal links. For

every action $a \in \mathcal{P}$, and for every condition p of that action, we extract a candidate causal link set. The method GetCausalLinkCandidates returns such a set of candidates given such an a and p. During run time when $\mathcal{P}$ is dispatched, exactly one candidate causal link from every candidate causal link set will be *activated*, meaning that its associated predicate will be monitored. This guarantees that we will monitor all relevant conditions in our plan. Our approach to extracting these candidate causal links is based on reasoning over the conditions and effects of each action, as well as the partial ordering over those actions as defined by the action graph. The core offline algorithm PreprocessPlan returns as output a set of candidate causal link sets for subsequent use by the online stage.

The online suite of algorithms is responsible for dispatching the events, activating causal links, and monitoring those activated causal links. The scheduler and dispatcher once again make use of the APSP formulation, constructing a minimal dispatchable network in order to select time points for each event. As events are dispatched, the execution monitor updates the candidates from the candidate causal link sets, until it can be sure that a causal link should be activated. Once activated, the causal link will be monitored continuously until it is deactivated and no longer relevant. When no causal links are violated, we can guarantee that the current execution trace is healthy. Furthermore, should a causal link be violated, we can also guarantee that the execution trace is unhealthy. If such an unhealthy trace is detected, Pike returns a conflict in the form of a minimal set of constraints that cannot simultaneously hold true.

Now that we have provided a high level overview of our algorithmic approach and how these pieces fit together, the following sections will dive into greater detail in each of these algorithms. Each section will provide further details, pseudo code, and proofs of various guarantees made by these algorithms. Once we have finished presenting our solution to the plan executive, this chapter will conclude with theoretical and empircal analyses of these algorithms.

2.4.2 Partial Ordering over Actions

Although it is not generally possible to construct a total ordering of actions in a plan, it is crucial for the execution monitor's causal link extraction algorithms to make use of a partial ordering of actions in a plan. This section details the process through which we take a temporally flexible plan $\mathcal{P}$ and derive such a partial ordering, which we store in the *action graph*.

Intuitively, we say that some action in a plan *precedes* some other action if the first action is guaranteed to finish before the second action starts. Put another way, the first action's finish event must precede the second action's start event in all consistent schedules. In general for any two actions a_1 and a_2, there are three possibilities: 1.) a_1 may precede a_2, 2.) a_2 may precede a_1, xor 3.) neither a_1 and a_2 precedes the other. In this case a_1 and a_2 may overlap temporally, or a precise ordering cannot be determined in the absence of a schedule, and so we cannot say that either action precedes the other. We refer to this last case as the actions being *incomparable*.

This precedence relation can be formalized as a partial ordering over the actions in a plan as follows:

Definition 2.4.1 (Partial Ordering of Actions in a Temporal Plan)**.** Given a temporal plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ and any two actions $a_1, a_2 \in \mathcal{A}$, we say that a_1 precedes a_2 or $a_1 \prec a_2$ if a_1 finishes at or before the time that a_2 starts. Formally, letting $a_1 = \langle \boldsymbol{\alpha_1}, e_{s_1}, e_{f_1} \rangle$ and $a_2 = \langle \boldsymbol{\alpha_2}, e_{s_2}, e_{f_2} \rangle$, we say $a_1 \prec a_2$ iff $t_{e_{f_1}} \leq t_{e_{s_2}}$ in all consistent schedules of $\mathcal{P}$. In general, exactly one of the following will hold: 1.) $a_1 \prec a_2$, 2.) $a_2 \prec a_1$, xor 3.) $a_1 \parallel a_2$ (a_1 and a_2 are incomparable - it is unknown whether one action will precede the other without being given a specific schedule).

There are several useful facts about this partial ordering that are useful and will be built upon later in this thesis. We introduce them below:

Lemma 2.4.1. *The $\prec$ relation is transitive: $a_1 \prec a_2 \bigwedge a_2 \prec a_3 \Rightarrow a_1 \prec a_3$.*

Lemma 2.4.2. *The $\parallel$ relation is not transitive: $a_1 \parallel a_2 \bigwedge a_2 \parallel a_3 \not\Rightarrow a_1 \parallel a_3$.*

Lemma 2.4.3. *For any action a, $a \parallel a$ holds true.*

Lemma 2.4.4. *For any pair of actions a_1 and a_2, it cannot be the case that $a_1 \prec a_2 \bigwedge a_2 \prec a_1$. Intuitively, this is not possible because we constrain each action to have positive (> 0) duration.*

When we have a fixed schedule for a plan, it is possible to determine a total ordering over actions within the plan by evaluating the criteria above with the assigned time values for each event. The above definition however holds true not for some fixed schedule, but for all consistent schedules in a plan. It is designed so that, during the offline stage when a precise schedule has not yet been determined, we can still deduce ordering relations between some pairs of events. This is depicted visually in Figure 2-12.

The causal link extraction algorithms that follow require the ability to query this partial order relation for any pair of actions in the plan. As such, we choose to represent the precedence relations in graph structure we call the action graph for a plan:

Definition 2.4.2 (Action Graph)**.** The *action graph* for a plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ is a graph $G = (V, E)$ such that $V = \mathcal{A}$, and there is an edge $(a_1, a_2) \in E$ if and only if $a_1 \prec a_2$.

The action graph also satisfies the following properties:

Corollary (Action Graph Edges)**.** *For every pair of actions in the action graph, exactly one of the following will hold true:*

- *An edge (a_1, a_2) will exist, representing $a_1 \prec a_2$*
- *An edge (a_2, a_1) will exist, representing $a_2 \prec a_1$*
- *No edge between a_1 and a_2 will exist, representing that $a_1 \parallel a_2$.*

The algorithm ConstructActionGraph (shown in Algorithm 1) constructs this action graph. After initialization, the algorithm first generates a distance graph from the plan $\mathcal{P}$ and proceeds to apply the Floyd Warshall all-pairs shortest path algorithm ([12]). This returns a look up table $dist(e_1, e_2)$ valid over any e_1, e_2 in $\mathcal{P}$'s

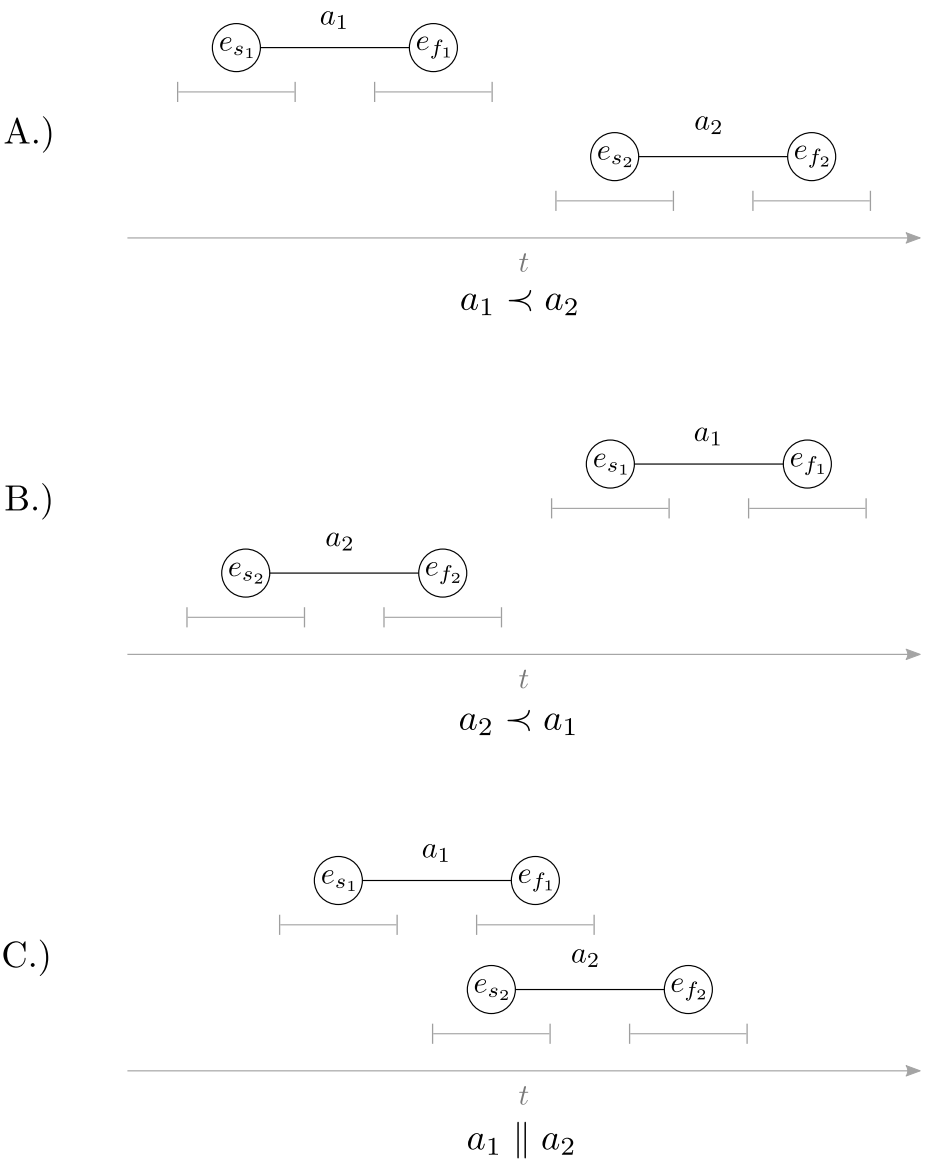

Figure 2-12: Illustrations of the various partial order precedence relations for plans with temporal flexibility. The actions a_1 and a_2 are shown, each with their corresponding start and finish events. Time flows from left to right, and the gray bar underneath each event indicates the range of valid scheduled times for that event as computed by the all pairs shortest path algorithm. In A.) we see that $t_{e_{s_2}} \geq t_{e_{f_1}}$ in all possible event timings, so it is guaranteed that $a_1 \prec a_2$ in all schedules. The results are similar in B.), except reversed where now $a_2 \prec a_1$. In C.) we see that we cannot guarantee that a_1 will finish before a_2 starts. While this may occur in some schedules, it is will not occur in all of them. Therefore the ordering of a_1 and a_2 cannot be determined before the plan is scheduled, so we say that $a_1 \parallel a_2$.

Algorithm 1: ConstructActionGraph

Data: $\mathcal{P} =< \mathcal{E}, \mathcal{C}, \mathcal{A} >$

Result: An action graph $G = (V, E)$, or Failure.

```
1  begin
2      V ⟵ A
3      E ⟵ ∅
4      distance-graph ⟵ DistanceGraph(P)
5      dist(·,·) ⟵ FloydWarshall(distance-graph)
6      if dist(e,e) < 0 for any e ∈ E then
7          return Failure
8      end
9      foreach a = ⟨α, e_s, e_f⟩ ∈ A do
10         if dist(e_f, e_s) ≥ 0 then
11             return Failure
12         end
13     end
14     foreach a_1 = ⟨α_1, e_{s_1}, e_{f_1}⟩ ∈ A do
15         foreach a_2 = ⟨α_2, e_{s_2}, e_{f_2}⟩ ∈ A do
16             if dist(e_{s_2}, e_{f_1}) ≤ 0 then
17                 E = E ∪ (a_1, a_2)
18             end
19         end
20     end
21 end
```

events. A value $dist(e_1, e_2) = d$ represents the computed explicit constraint that, in all consistent schedules, $t_{e_2} - t_{e_1} \leq d$.

The first loop in the algorithm checks if $dist(e_f, e_s) \geq 0$ for any start and event times for an action a. If this is so, it represents that in some consistent schedules it is possible that $t_{e_s} - t_{e_f} \leq d$ for some $d \geq 0$, indicating that the action may take 0 or negative duration. This is disallowed, so FAILURE is returned if this condition is detected.

The nested "for" loops that iterate over pairs of actions are responsible for adding edges to the graph. If it is the case that for some pair of actions $a_1 = \langle \boldsymbol{\alpha_1}, e_{s_1}, e_{f_1} \rangle$ and $a_2 = \langle \boldsymbol{\alpha_2}, e_{s_2}, e_{f_2} \rangle$ that $dist(e_{s_2}, e_{f_1}) \leq d$ for some $d \leq 0$, then we can rewrite the associated constraint as $t_{e_{s_2}} - t_{e_{f_1}} \geq -d$. Since $-d \geq 0$, this constraint tells us that the start event of the a_2 must occur at or after the finish event of a_1, and so hence $a_1 \prec a_2$ and we add an edge in the graph. When repeated for all pairs of action, this constructs the complete action graph.

The action graph is used to evaluate $\prec$ in all following discussion and algorithms. Although not explicitly represented in any of the pseudo code in algorithms to come, whenever we write $a_1 \prec a_2$ for some pair of actions we imply a query to the action graph to check if an edge (a_1, a_2) exists.

2.4.3 Causal Links

In this section, we formalize the notion of causal links, which are at the core of our execution monitoring algorithm.

Intuitively, a *causal link* captures the notion that "action a_1 is the producer of predicate p, which is required for action a_2 to run" [25]. We further enforce that no other action may *threaten* the causal link by negating p at some point in time between a_1 and a_2. If some action were to do so, presumably p would not hold true by a_2's start time, and hence the preconditions of the action would be violated.

Building upon the definition in [20], we define a causal link as follows:

Definition 2.4.3 (Causal Link)**.** A *causal link* for a plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ is a tuple

$\langle a_p, p, a_c \rangle$, denoted $a_p \xrightarrow{p} a_c$ where $a_p = \langle \boldsymbol{\alpha_p}, e_{s_p}, e_{f_p} \rangle$ and $a_c = \langle \boldsymbol{\alpha_c}, e_{s_c}, e_{f_c} \rangle$ are both actions in $\mathcal{A}$ and p is an instantiated PDDL predicate. We call a_p the *producing action* and a_c the *consuming action*. The following requirements must hold for $a_p \xrightarrow{p} a_c$:

- The producing action produces p as one of it's effects: $p \in \textsc{Effects}(\boldsymbol{\alpha_p})$.
- The consuming action uses p as one of its preconditions: $p \in \textsc{Precond}(\boldsymbol{\alpha_c})$.
- The producing action precedes the consuming action: $a_p \prec a_c$
- There is no action $a_{threat} = \langle \boldsymbol{\alpha_{threat}}, e_{s_{threat}}, e_{f_{threat}} \rangle$ that *threatens* the causal link. We say an action a_{threat} *threatens* the causal link if $\neg p \in \textsc{Effects}(\boldsymbol{\alpha_{threat}})$ and any of the following hold true:
  - $a_p \prec a_{threat} \prec a_c$
  - $a_p \parallel a_{threat}$
  - $a_c \parallel a_{threat}$

Please note that, in the above definition, some action can threaten the causal link if it is incomparable with either the producer or the consumer actions. This has the implication that the above definition of causal link is conservative in the sense that actions deemed as threats may not actually be problematic in all schedules. For example, consider some threat a_{threat} that produces $\neg p$ and is incomparable with a_c. In some schedules, it may turn out to be the case that a_{threat} actually starts after a_c ends. However, since this determination cannot be made in general for all schedules, we mark a_{threat} as a potential threat. We are concerned with *all consistent schedules*, so if $a_p \parallel a_{threat}$ or $a_c \parallel a_{threat}$ then we cannot guarantee beforehand that p will not be threatened. Similarly, we also require that $a_p \prec a_c$, or namely that a_p must finish before a_c in *all* possible consistent schedules of the plan.

In addition to a causal link, we also introduce the notion a *candidate causal link*. A *candidate causal link* meets the definition of a causal link, but has the additional restriction that it has one of the "latest occurring" producing actions a_p. Intuitively, this is useful because it is the latest-occurring producing action that is the cause of

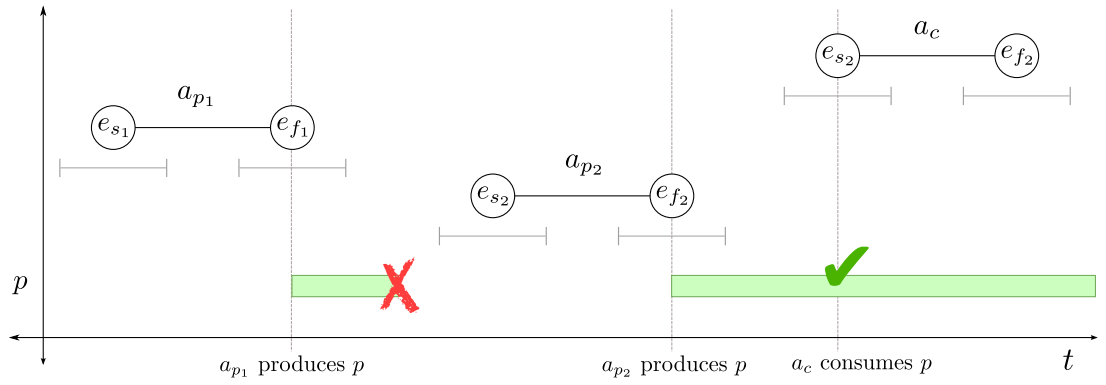

Figure 2-13: This figure illustrates the importance of candidate causal links, as differentiated from causal links. Suppose we have two candidate producer actions, a_{p_1} and a_{p_2}. Which would make a better producer in a causal link? Both produce p at their finish times. However, since $a_{p_1} \prec a_{p_2}$, we can guarantee that a_{p_2} producers p later than a_{p_1} in all schedules. Therefore, even if a disturbance happens and p is negated (shown by the red X) early on, a_{p_2} will still be there to save the day and produce p for a_c. Therefore, it is most desirable to choose the latest occurring producer action. We would call $a_{p_2} \xrightarrow{p} a_c$ a candidate causal link, but $a_{p_1} \xrightarrow{p} a_c$ cannot be considered a candidate causal link since a_{p_2} will occur after a_{p_1}.

p being TRUE, not some earlier-occurring action. Suppose that for some consuming action a_c and some predicate p, there are two possible producing actions a_{p_1} and a_{p_2} that yield p as an effect. We thus can pick from two otherwise identical causal links: $a_{p_1} \xrightarrow{p} a_c$ and $a_{p_2} \xrightarrow{p} a_c$. Since $a_{p_1} \prec a_{p_2}$, we can intuitively think of a_{p_1} as being dominated by a_{p_2}. This is because in any consistent schedule, we know that a_{p_2} must produce p after a_{p_1}. As such it is irrelevant if a_{p_1} produces p earlier during execution. Please see Figure 2-13 for an example that shows this visually.

Definition 2.4.4 (Causal Link Dominance)**.** We say that $a_{p_2} \xrightarrow{p} a_c$ *dominates* $a_{p_1} \xrightarrow{p} a_c$ if a_{p_2} finishes after a_{p_1}. Formally, letting $a_{p_2} = \langle \boldsymbol{\alpha_2}, e_{s_2}, e_{f_2} \rangle$ and $a_{p_1} = \langle \boldsymbol{\alpha_1}, e_{s_1}, e_{f_1} \rangle$,

We argue that the *candidate causal link* is more useful than the generic causal link, and so the algorithms we present in this thesis find only sets of candidate causal links. We define a candidate causal link formally as follows:

Definition 2.4.5. A *candidate causal link* is a causal link such that no other action in the plan that produces p is guaranteed to come after a_p but before a_c. Formally, l is a candidate causal link if l is a causal link $a_p \xrightarrow{p} a_c$, and no other action $a' = \langle \boldsymbol{\alpha'}, e'_s, e'_f \rangle \in \mathcal{A}$ that has $p \in \textsc{Effects}(\boldsymbol{\alpha'})$ satisfies $a_p \prec a' \prec a_c$.

Please note that, according to this definition, there still may be multiple causal links for a given a_c and condition p. For all of these to be candidate causal links however, it must be the case that all of their producing actions are mutually incomparable. If this were not the case, then at least one would not be a candidate causal link. Our algorithms extract sets of candidate causal links during the preprocessing phase. At run time, once scheduling information is known and some events have been dispatched, we gain more information as to which producers actually occur the latest in time. We use this to always activate the latest candidate causal link.

2.4.4 Offline Causal Link Extraction Algorithm

Now that we have introduced the necessary formalisms and defined causal links and the action graph, we proceed to describe the core offline algorithms. The job of these algorithms is to extract sets of candidate causal links for the conditions of actions in the plan. These candidate causal links provide predicates and the time intervals over which they must be monitored online.

Our approach is to extract a set of candidate causal links $\mathcal{L} = \{l_1, l_2, ...\}$ for each condition of each action in $\mathcal{P}$, where each l_i is a candidate causal link. During online execution, exactly one l_i from each $\mathcal{L}$ will be activated and monitored.

In all of the discussions that follows, we make the assumption that the plan $\mathcal{P}$ has two distinguished actions, called a_{start} and a_{end}. The action a_{start} has no preconditions, and has effects that produce the initial world state. Similarly, a_{end} has no effects but has preconditions equal to the goal conjunction that was used to construct $\mathcal{P}$. This simplifies the causal link extraction process by allowing candidate causal links with producer action a_{start} should some predicate be true because of the initial conditions (and not because of some other action). Similarly, a candidate causal link

whose consumer action is a_{end} means that a predicate must hold true not for some later action to execute, but for the goal condition to be met.

Algorithm 2: PREPROCESSPLAN

Data: A plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ with distinguished start and end actions, a set of PDDL operator schema $\{\mathcal{O}_1, \mathcal{O}_2, ...\}$

Result: A mapping $\mathcal{D}$ of events to sets of candidate causal link sets, or FAILURE if the plan is not executable under $\mathcal{W}_{ideal}$

```
1  begin
2      G ⟵ ConstructActionGraph(P)
3      ComputeGroundedConditionsAndEffects(P)
4      D[e] ⟵ ∅ for each e ∈ E
5      foreach a_c = ⟨α_c, e_s, e_f⟩ ∈ A do
6          foreach p ∈ Precond(α_c) ∪ Maintenance(α_c) do
7              L ⟵ GetCausalLinkCandidates(p, a_c)
8              if L = ∅ then
9                  return Failure
10             else
11                 foreach l = ⟨α_p, e_sp, e_fp⟩ --p--> ⟨α_c, e_sc, e_fc⟩ ∈ L do
12                     Add L to D[e_fp]
13                 end
14             end
15         end
16     end
17     return D
18 end
```

The core preprocessing algorithm, PREPROCESSPLAN, is shown in Algorithm 2. The algorithm begins by computing the action graph, thereby allowing the $\prec$ relation to be evaluated in all subsequent methods. Next, the preconditions, maintenance conditions, and effects of each action $a \in \mathcal{A}$ are computed. Namely, a binding is computed and then applied to each of the preconditions, maintenance conditions, and effects. This allows the evaluation of such queries as $p \in \text{PRECOND}(\boldsymbol{\alpha})$ in subsequent methods.

The bulk of PREPROCESSPLAN is a set of nested loops that call GETCAUSALLINKCANDIDATES for every action a_c and condition p of that a_c. GETCAUSALLINKCANDIDATES is the main routine that does the bulk of the work in extracting sets of candidate causal links, and so most of our attention will focus on this function.

Algorithm 3: GetCausalLinkCandidates

Data: An action $a_c = \langle \boldsymbol{\alpha_c}, e_{s_c}, e_{f_c} \rangle$, a PDDL predicate $p \in \text{Precond}(\boldsymbol{\alpha_c})$, an action graph $G = (V, E)$ in order to evaluate precedence $\prec$, and a plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$

Result: A set of candidate causal links $\mathcal{L} = \{a_p \xrightarrow{p} a_c, ...\}$, or the empty set $\emptyset$ if no candidate causal links exist.

```
1  begin
2      ξ ⟵ ∅
3      foreach a ∈ A do
4          if a ≠ a_c and a ∥ a_c and ¬p ∈ Effects(α) then
5              return Failure
6          else if either p or ¬p ∈ Effects(α), and a ≺ a_c then
7              UpdateProducerCandidates(ξ, a)
8          end
9      end
10     foreach a ∈ ξ do
11         if ¬p ∈ Effects(α) then
12             return Failure
13         end
14     end
15     return {a_p --p--> a_c for each a_p ∈ ξ}
16 end
```

Algorithm 4: UpdateProducerCandidates

Data: A list of candidate actions ξ, a new action a_{new} to be processed

Result: Modifies ξ to include the new producing action candidate, if appropriate.

```
1  begin
2      foreach a ∈ ξ do
3          if a_new ≺ a then
4              return
5          end
6      end
7      foreach a ∈ ξ do
8          if a ≺ a_new then
9              Remove a from ξ
10         end
11     end
12     Add a_new to ξ
13 end
```

The algorithm GetCausalLinkCandidates is shown in Algorithm 3. The input to this method is a temporal plan $\mathcal{P}$, an action a_c, and a predicate p that is a precondition or maintenance condition of a_c. The output of GetCausalLinkCandidates is a set $\mathcal{L}$ of candidate causal links that all have consumer a_c and predicate p. If the algorithm finds no candidate causal links and hence returns the empty set $\emptyset$, this means that all schedules of $\mathcal{P}$ are not executable under $\mathcal{W}_{ideal}$.

GetCausalLinkCandidates operates by building up a set of producer actions, which represent the set of possible producers for the candidate causal links returned in $\mathcal{L}$. These producer candidates are stored in the set ξ. Intuitively, GetCausalLinkCandidates iterates over all actions $a \in \mathcal{A}$. It continually updates ξ with actions using the subroutine UpdateProducerCandidates, shown in Algorithm 4. After processing each action, ξ contains the largest set of possible producer actions that are all mutually incomparable and "closest" to a_c (but preceding it). By ensuring that all of the producer candidates are all as temporally close to a_c as possible and mutually incomparable, we ensure that that each of the causal links returned in $\mathcal{L}$ will in fact be a candidate causal link.

Properties of Candidate Causal Link Extraction Algorithms

In this section, we proceed to outline and prove various useful properties about GetCausalLinkCandidates and its UpdateProducerCandidates. We will then prove that GetCausalLinkCandidates is a sound and complete algorithm.

We begin by proving an important invariant condition of ξ throughout execution. Namely, after each call to UpdateProducerCandidates(ξ, a_{new}), all elements of ξ are mutually incomparable.

Theorem 2.4.5 (Mutual Incomparability of ξ)**.** *For every pair of elements* $a_1, a_2 \in \xi$, $a_1 \parallel a_2$.

Proof. Proof by Invariance. There are two base cases: when ξ is empty and when it contains a single element. The base case where $\xi = \emptyset$ makes our claim vacuously true. When $\xi = \{a\}$ for some a, the claim holds because any action is always incom-

parable to itself. Now that we have demonstrated that the base cases hold, we prove the invariant also holds after each call to UpdateProducerCandidates(ξ, a_{new}) (which may possibly grow a_{new}), thereby proving the claim.

Suppose that UpdateProducerCandidates(ξ, a_{new}) is called. We assume by our invariant condition that that all elements $a \in \xi$ are mutually incomparable. Therefore, since a_{new} is the only possible addition to ξ, we need only check the incomparability between the new element a_{new} and each of the elements in ξ. If $a_{new} \prec a$ for any $a \in \xi$, lines 2-6 enforce that a_{new} will not be added and hence ξ will remain unaltered, so the invariant will hold as UpdateProducerCandidates returns. Lines 7-11 enforce that, should $a \prec a_{new}$ for any $a \in \xi$, then a will be removed from ξ. Thus, since neither $a_{new} \prec a$ nor $a \prec a_{new}$ can hold for any $a \in \xi$, it is safe to say that $a_{new} \parallel a$ for all $a \in \xi$. Thus the invariant will hold after UpdateProducerCandidates returns. □

Next, we proceed to prove that, after iterating over each $a \in \mathcal{A}$, GetCausalLinkCandidates contains all actions that are "closest" to a_c and affect p by either asserting p or $\neg p$. Put another way, there are no other actions that affect p and are guaranteed to come closer to a_c in all schedules of the plan.

Lemma 2.4.6. *Upon reaching line 10 in the* GetCausalLinkCandidates *algorithm, there is no action that affects p and comes "closer" to a_c than any of the actions in ξ. In other words, there is no $a_{closer} \in \mathcal{A}$ but not in ξ such that p or $\neg p \in$ Effects(a_{closer}) and $a_{cand} \prec a_{closer} \prec a_c$ for any $a_{cand} \in \xi$.*

Proof. Proof by Contradiction. Suppose that there does exist some action $a_{closer} \in \mathcal{A}$ but not in ξ such that p or $\neg p \in$ Effects(a_{closer}), and $a_{cand} \prec a_{closer} \prec a_c$ for some action $a_{cand} \in \xi$. Since the loop in lines 3-9 of GetCausalLinkCandidates iterates over each action in $\mathcal{A}$ and a_{closer} meets the else-if requirement, then UpdateProducerCandidates(ξ, a_{closer}) must have been called in an attempt to add a_{closer} to ξ, but a_{closer} did not actually get added or did not remain in ξ. There are two possibilities - either this attempted-addition occurred before a_{cand} was added, or after a_{cand} was added. We consider both of these scenarios below:

1. Suppose that a_{closer} was attempted to be added before a_{cand} was added to ξ. If $a_{cand} \prec a_{closer}$ then it certainly would have been added to ξ. But since $a_{cand} \prec a_{closer}$, a_{cand} would not be added to ξ later on. This contradicts our assumption that $a_{cand} \in \xi$.

2. Suppose that a_{cand} was added to ξ before the attempted addition of a_{closer}. Then, since $a_{cand} \prec a_{closer}$, a_{cand} would be purged from ξ upon the addition of a_{closer}. This also contradicts our assumption that $a_{cand} \in \xi$.

Therefore, since in either case we reach a contradiction, it must be the case that there does exist some action $a_{closer} \in \mathcal{A}$ but not in ξ such that p or $\neg p \in \textsc{Effects}(a_{closer})$, and $a_{cand} \prec a_{closer} \prec a_c$ for an action $a_{cand} \in \xi$. □

Now that we have proven the above two useful properties, we proceed to demonstrate that GetCausalLinkCandidates is sound. Every element of $\mathcal{L}$ is a candidate causal link.

Theorem 2.4.7 (GetCausalLinkCandidates Soundness)**.** GetCausalLinkCandidates *is sound. In other words, If* $p \in \textsc{Precond}(\boldsymbol{\alpha_c}) \cup \textsc{Maintenance}(\boldsymbol{\alpha_c})$*, then every element in the list* $\mathcal{L} = \textsc{GetCausalLinkCandidates}(p, a_c)$ *is a candidate causal link.*

Proof. We need to prove that each of the $l = a_p \xrightarrow{p} a_c \in \mathcal{L} = \textsc{GetCausalLinkCandidates}()$ meets the definition of a candidate causal link (Definition 2.4.5). Namely, we prove the following that for each l: 1.) $p \in \textsc{Effects}(\boldsymbol{\alpha_p})$, 2.) $a_p \prec a_c$, 3.) No actions threaten the causal link, and 4.) There are no other actions a' that have $p \in \textsc{Effects}(\boldsymbol{\alpha'})$ where $a_p \prec a' \prec a_c$. Proving all of these claims is sufficient to show that l is a candidate causal link.

We begin by proving that $p \in \textsc{Effects}(\boldsymbol{\alpha_p})$. We note that a_p is always drawn from ξ, which contains only actions that produce either p or $\neg p$ (line 6). The loop in lines 10 - 14 will return if any actions in ξ produce $\neg p$. Therefore, when a random a_p is selected it is guaranteed to produce p as an effect.

Using similar logic, we can see that elements can only be added to ξ should the condition $a_p \prec a_c$ be true (line 6). Therefore we can be sure that $a_p \prec a_c$ in any causal link returned.

Next, we ensure that there are no actions that would threaten the causal link. We see on line 6 that no causal link can be returned should a threat a_{threat} be detected such that $\neg p \in \textsc{Effects}(\boldsymbol{\alpha_{threat}})$ and $a_{threat} \parallel a_c$. Additionally, the loop in lines 10 - 14 ensures that no causal link can be returned where such a threat action would be incomparable to the producing action, i.e. $a_{threat} \parallel a_p$. The final step remaining is to show that there is no a_{threat} with $\neg p \in \textsc{Effects}(\boldsymbol{\alpha_{threat}})$ such that $a_p \prec a_{threat} \prec a_c$. This can be shown using Lemma 2.4.6. Since a_p is drawn from ξ, and there can be no action $a_{closer} \in \mathcal{A}$ that affects p and that satisfies $a_{cand} \prec a_{closer} \prec a_c$, we can be sure that no threat could possibly come after a_p.

Thus far, we have proven that l is a causal link. We finish by proving that l is a candidate causal link, or namely that there is no other action a' that has $p \in \textsc{Effects}(\boldsymbol{\alpha'})$ where $a_p \prec a' \prec a_c$. By Lemma 2.4.6, there can be no action a_{closer} that affects p where $a_{cand} \prec a_{closer} \prec a_c$ for any element $a_{cand} \in \xi$. Therefore no such a' can exist, otherwise the candidate causal link would be $a' \xrightarrow{p} a_c$.

The above conditions satisfy all of the conditions set forth in the definition of a candidate causal link (Definition 2.4.5). Hence, GetCausalLinkCandidates is sound. □

We conclude this section with our final proof and show that in addition to being sound, GetCausalLinkCandidates is also complete. That is, it returns all possible candidate causal links for a_c and p.

Theorem 2.4.8 (GetCausalLinkCandidates Completeness)**.** GetCausalLinkCandidates *is complete. In other words,* GetCausalLinkCandidates(p, a_c) *returns all possible candidate causal links for* a_c *and* p*.*

Proof. Proof by contradiction. Suppose that GetCausalLinkCandidates is not complete, and so there must be some candidate causal link $a' \xrightarrow{p} a_c$ that exists but is not in the returned set $\mathcal{L}$. The producing actions for the candidate causal links

returned in $\mathcal{L}$ are generated using the producer candidates in ξ. Thus, $a' \notin \xi$ since $a' \xrightarrow{p} a_c$ is not in $\mathcal{L}$.

We will consider three different scenarios, one of which must hold true and each of which leads to a contradiction. One of the following must hold true: 1.) $a_p \prec a'$ for some $a_p \in \xi$, 2.) $a' \prec a_p$ for some $a_p \in \xi$, or 3.) $a_p \parallel a'$ for all $a_p \in \xi$.

1. In the first case, $a_p \prec a'$ for some $a_p \in \xi$. However, this contradicts Lemma 2.4.6, which states that no action $a' \notin \xi$ that affects p may satisfy $a_p \prec a' \prec a_c$ for all $a_p \in \xi$.

2. In the second case, $a' \prec a_p$ for some $a_p \in \xi$. Then $a' \xrightarrow{p} a_c$ cannot be a *candidate* causal link (though it is a causal link), contradicting our assumption.

3. In the final case, since neither $a_p \prec a'$ nor $a' \prec a_p$ for any element $a_p \in \xi$, we can be sure that $a_p \parallel a'$ for all a_p. Since ξ contains the largest set of incomparable elements closest to ξ, then $a' \in \xi$. This contradicts our assumption however that $a' \notin \xi$.

Each of the three cases above leads to a contradiction, so GetCausalLinkCandidates must return all possible candidate causal links and is hence complete. □

2.4.5 Online Plan Execution and Monitoring Algorithms

Thus far, all algorithmic discussion has focused on the offline preprocessing stage responsible for extracting sets of candidate causal links for each predicate of each action of the plan. In this section, we discuss the online algorithms used to implement execution monitoring. We formalize the Pike plan executive, focusing on the online dispatcher that is tightly integrated with the execution monitor. We also prove some strong guarantees about the execution monitor. Namely, we show that by monitoring activated causal links, the execution monitor will be be able to efficiently deduce whether the current execution trace is healthy or not.

We begin by presenting the Pike dispatcher, shown in Algorithm 5. This algorithm is responsible for scheduling events and dispatching activities from a plan online, while

simultaneously running the execution monitor. This stage makes use of a minimal dispatchable network computed offline ([22]). This minimal dispatchable network is computed by running an all pairs shortest path algorithm (APSP), and subsequently removing unnecessary dominated edges. When complete, the minimal dispatchable network allows Pike to efficiently query and update the APSP temporal windows for each event, a step that is key to online event dispatching. It has been proven that given an execution trace for a plan where each event time assignment falls within their APSP temporal windows, there exists a temporally consistent extension to the plan that is temporally feasible (([5]).

Pike also makes the assumption that it does not explicitly schedule the times for action finish events. Rather, once an the start event for an action is dispatched, the plant will choose the precise finish event time for that action. Should this action finish event occur outside of it's APSP temporal window as dictated by the minimal dispatchable network, Pike returns a conflict since no extension to the execution trace can be temporally feasible. In this situation, a conflict containing temporal-related constraints is returned. The PIKE algorithm maintains a set S of dispatched events and a set $\mathcal{A}_{active}$ of events that are currently in progress to keep track of dispatched events and currently active activities for which the finish events have not yet executed.

The PIKE algorithm also has hooks that call key algorithms of the execution monitor: INITEXECMON, EXECMONEVENTUPDATE, and ISEXECUTIONGOOD?. These algorithms will be described in great detail shortly. From a high level, INITEXECMON is responsible for initializing the active monitors, EXECMONEVENTUPDATE processes events being dispatched in order to activate or deactivate monitor conditions corresponding to causal links or action maintenance conditions, and ISEXECUTIONGOOD? monitors the set of current monitor conditions to ensure that all predicates hold TRUE. If a monitor condition is violated (and hence an activated causal link or maintenance condition is violated), then Pike returns immediately.

Algorithm 5: PIKE

Data: A plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$, a mapping $\mathcal{D}$ from events to sets of candidate causal link sets

Result: A proper dispatch of all events in $\mathcal{E}$ at correct times, a minimal conflict if failure occurs.

```
1  begin
2      InitExecmon(L)
3      S ⟵ ∅
4      A_active ⟵ ∅
5      while E \ S ≠ ∅ do
6          foreach action a = ⟨α, e_s, e_f⟩ ∈ A_active that just reported completion do
7              A_active = A_active \ {a}
8              S = S ∪ e_f
9              ExecmonEventUpdate(e_f)
10         end
11         E_cand = all events e ∈ E \ S within their APSP temporal windows now
12         E_cand = E_cand \ {any e_f of any action a = ⟨α, e_s, e_f⟩ ∈ A_active}
13         foreach e ∈ E_cand do
14             S = S ∪ e
15             if e = e_s for some a = ⟨α, e_s, e_f⟩ ∈ A then
16                 A_active = A_active ∪ {a}
17                 DispatchAction(α)
18                 ExecmonEventUpdate(e_s)
19             end
20         end
21         if any e ∈ E \ S is past its window of execution then
22             Error(Temporal infeasibility)
23             return Conflict({e's temporal window, t = t_now})
24         end
25         ν ⟵ IsExecutionGood?()
26         if ν ≠ ∅ then
27             Error(Activated causal links violated! Replanning needed.)
28             return ν
29         end
30     end
31 end
```

Activated Causal Links and Monitor Conditions

Before we dive into these execution monitoring algorithms, we first introduce several concepts related to causal links. Specifically, we formally define several important properties of activated causal links, and introduce the notion of a *monitor condition*.

We begin my defining an *activated causal link*. Given a set $\mathcal{L}$ of candidate causal links, the activated causal link of this set is the causal link with the latest-occurring producer finish event.

Definition 2.4.6 (Activated Causal Link)**.** The *activated causal link* l of a candidate causal link set $\mathcal{L}$ is the single candidate causal link in $\mathcal{L}$ that has the latest-scheduled producer finish event. It is the causal link whose producing action finishes the latest in time. Formally, given a set of candidate causal links $\mathcal{L}$, the activated causal link of $\mathcal{L}$ is the candidate causal link $l = \langle \boldsymbol{\alpha_p}, e_{s_p}, e_{f_p} \rangle \xrightarrow{p} \langle \boldsymbol{\alpha_c}, e_{s_c}, e_{f_c} \rangle \in \mathcal{L}$ such that $t_{e_{f_p}} \geq t_{e'_{f_p}}$ for all other $l' = \langle \boldsymbol{\alpha'_p}, e'_{s_p}, e'_{f_p} \rangle \xrightarrow{p'} \langle \boldsymbol{\alpha'_c}, e'_{s_c}, e'_{f_c} \rangle \in \mathcal{L}$. In the case of a tie where two candidate causal links have precisely the same producer finish time, we choose one arbitrarily.

We proceed by defining a related concept, the *activation window* for a causal link. Intuitively, the activation window is the temporal duration during which a causal link's predicate must be monitored.

Definition 2.4.7 (Activation Window for a Causal Link)**.** We say that the *activation window* for a causal link l is the temporal duration starting from l's producer finish event and ending at l's consumer start event. Formally, the *activation window* for a causal link $l = \langle \boldsymbol{\alpha_p}, e_{s_p}, e_{f_p} \rangle \xrightarrow{p} \langle \boldsymbol{\alpha_c}, e_{s_c}, e_{f_c} \rangle$ is the temporal duration $t_{e_{f_p}} \leq t \leq t_{e_{s_c}}$. Furthermore, we say that a candidate causal link is *activated* at time $t_{e_{f_p}}$ and *deactivated* at time $t_{e_{s_c}}$.

The execution monitor algorithm activates causal links at the producer event's finish time, and deactivates them at the consumer event's start time. During this interval, the predicate p must be monitored. Should we find that at some time p does not hold TRUE, we say that the causal link is *violated*:

Definition 2.4.8 (Valid/Violated Causal Link)**.** We say that a causal link is *valid* at some time if its predicate is true at that time. Otherwise, if the predicate is false, we say that the causal link is *violated*. Formally, a causal link $l = a_p \xrightarrow{p} a_c$ is *valid* under $\mathcal{W}$ at time t if $p \in \mathcal{W}(t)$. Otherwise, we say that l is *violated* under $\mathcal{W}$ at time t.

The concept of a violated causal link is central to the execution monitor. As will be proven shortly, should a causal link be violated, the execution trace is provably unhealthy.

In addition to causal links, we must also monitor other conditions during execution. The maintenance conditions of actions fall into this category. Like causal links, maintenance conditions require some predicate to be monitored continuously over some duration of time. Thus, in order to generalize the execution monitor, we introduce the notion of a *monitor condition*. Intuitively, a monitor condition contains start and end events from a plan, a predicate to be monitored during the interval between those start and end events, as well as a reason for being monitored. Both causal links and action maintenance conditions can be mapped to monitor conditions in a straightforward way. As such, the online monitor algorithms operate over monitor conditions to provide a clean formulation (as opposed to causal links and maintenance conditions dealt with separately). We define a monitor condition as follows:

Definition 2.4.9 (Monitor Condition)**.** A *monitor condition* is a condition that is monitored online during the duration between two event in a plan. Formally, a *monitor condition* with respect to a plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ is a tuple $\langle e_s, e_f, p, r \rangle$ where $e_s, e_f \in \mathcal{E}$, p is a PDDL predicate, and r is a reason. In our context, if r is a causal link, then the monitor condition represents a causal link whose predicate p is being monitored. If r is an action $a \in \mathcal{A}$, this represents a maintenance condition that is being monitored over the course of some action being executed.

Online Execution Monitoring Algorithms

Now that we have introduced all of the necessary formalisms, we describe the execution monitoring algorithms InitExecmon, ExecmonEventUpdate, and IsExecutionGood?. These online monitoring algorithms make use of a set $\mathcal{M}_{active}$. Each element $m \in \mathcal{M}_{active}$ is a monitor condition that must be continuously checked. InitExecmon, shown in Algorithm 6, simply clears $\mathcal{M}_{active}$ to the empty set. Throughout the course of execution when ExecmonEventUpdate is called as events are dispatched, monitor conditions are added and removed from $\mathcal{M}_{active}$ based on the causal links and maintenance conditions of actions. The IsExecutionGood? algorithm continuously loops over all $m \in \mathcal{M}_{active}$ to check their validity. We describe each of these algorithms in more detail in the following paragraphs.

ExecmonEventUpdate The ExecmonEventUpdate is called each time an event $e \in \mathcal{E}$ is dispatched (Algorithm 7). It is responsible for performing the following three tasks: 1.) Adding monitor conditions to $\mathcal{M}_{active}$ corresponding to causal links being activated, 2.) Adding monnitor conditions to $\mathcal{M}_{active}$ corresponding to maintenance conditions for actions, and 3.) Removing any monitor conditions from $\mathcal{M}_{active}$ that are no longer activated. The algorithm as shown executes these three steps sequentially.

Recall from the earlier discussion about candidate causal links that it is beneficial to know which producer actions finish the latest in time. It is this action that "causes" the consumer action's preconditions to be met. During the preprocessing phase, the best that we can do is generate a set of candidate causal links with mutually incomparable producer effects. However, during online execution, we have the additional information of when certain events have been dispatched. As a result, it is possible to wait for the latest producer event of all the causal link candidates for some $\mathcal{L}$. This guarantees that the candidate causal link that we activate is the one with the latest possible producer event, thereby ensuring that we monitor its associated predicate only when relevant. The ExecmonEventUpdate algorithm does exactly this. Each time that ExecmonEventUpdate is called with event e, it will remove

the candidate causal link from each set of candidate causal links that has a producer finish event e. If this was the last candidate causal link in the set, then we can be sure that this causal link has the latest producer action of all other causal links. The algorithm then activates that causal link.

The offline processing algorithms provide the information about candidate causal link sets in the form of $\mathcal{D}$, a mapping from events in the plan to sets of candidate causal link sets. For any event $e \in \mathcal{E}$, $\mathcal{D}[e] = \{\mathcal{L}_1, \mathcal{L}_2, ...\}$ such that e is the producer finish event for one of the causal links $l \in \mathcal{L}_i$. If some action produces multiple effects, then that action may be the producer in multiple causal links. This is why $\mathcal{D}[e]$ returns a set of candidate causal link sets (as opposed to a single candidate causal link set); each $\mathcal{L}_i \in \mathcal{D}[e]$ represents a different potentially-activated causal link for which e is the producer finish event. As noted above, the causal link l for which e is the producer finish event is removed from each $\mathcal{L}_i$. If any of these $\mathcal{L}_i$ are then empty after the removal, then we know that e is that latest-occurring producer finish event for all of the candidate causal links that were in $\mathcal{L}_i$. Therefore, the algorithm proceeds to convert the causal link l to a monitor condition m that is subsequently added to $\mathcal{M}_{active}$.

The next step in ExecmonEventUpdate is to create any monitor conditions corresponding to maintenance conditions that have now become activated. If the dispatched event e is the start event of any action in the plan, a monitor condition m is created for each maintenance condition that must be monitored in the plan, ending at the actions finish event. This m is then added to $\mathcal{M}_{active}$.

Finally, the last step in ExecmonEventUpdate is to remove any monitor conditions that have finished. If e is the finish event of any monitor condition m, then m is removed from $\mathcal{M}_{active}$. If m was generated from a causal link, this is equivalent to deactivating the causal link. If m were generated from a maintenance condition, this is equivalent to the action ending the maintenance condition no longer being required to hold True.

IsExecutionGood? We now describe the IsExecutionGood? method of the execution monitor, which is called continually at a high frequency fixed interval by the plan executive Pike algorithm. This method returns a set of conflicts if monitor conditions/causal links are violated, and otherwise returns the empty set $\emptyset$ if no violations are detected. The algorithm iterates over all monitor conditions $m \in \mathcal{M}_{active}$, checking to see if the predicate is True. The primitive routine IsTrue checks observations of the world state, as could be measured by a hybrid estimation module. Should any violation be detected, a minimal conflict in the form of the current execution trace, the observed negated predicate, and reason (either a causal link or a maintenance condition) are returned. This constitutes a minimal conflict, as removing any of these constraints would resolve any logical inconsistency.

Algorithm 6: InitExecmon

Data: A plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$, a set of causal link candidates $\mathcal{D}$
Result: Initializes the execution monitor system.

```
1 begin
2 |  M_active ⟵ ∅
3 end
```

Properties and Guarantees of the Execution Monitor

In this section, we prove various properties and guarantees about the execution monitor. These properties related to the proper activation of causal links and the conditions necessary for an execution trace to be healthy. We then proceed with two crucial proofs: 1.) we prove that if an activated causal link is violated, the execution trace is unhealthy, and 2.) we prove that if no activated causal link is violated, the execution trace is healthy.

Lemma 2.4.9. *During execution, exactly one candidate causal link will be activated for every condition p of every action a_c in the plan.*

Proof. The method GetCausalLinkCandidate is called for every condition p of every action a_c in the plan, and returns a set of candidate causal links $\mathcal{L}$, each with

Algorithm 7: EXECMONEVENTUPDATE

Data: An event e that is being dispatched, the mapping $\mathcal{D}$ from events to sets of candidate causal link sets

Result: An updated set of activated monitor conditions.

```
1  begin
2      foreach L ∈ D[e] do
3          Let l = the causal link in L whose producing action's finish event (e_{f_p})
           is e
4          L = L \ l
5          if L = ∅ then
6              Let l = ⟨α_p, e_{s_p}, e_{f_p}⟩ --p--> ⟨α_c, e_{s_c}, e_{f_c}⟩
7              Add ⟨e_{f_p}, e_{s_c}, p, l⟩ to M_active
8          end
9      end
10     if e = e_s for some a = ⟨α, e_s, e_f⟩ ∈ A then
11         foreach p ∈ MAINTENANCE(α) do
12             Add ⟨e_s, e_f, p, a⟩ to M_active
13         end
14     end
15     Remove from M_active any m = ⟨e_s, p, e_f⟩ such that e_f = e
16 end
```

Algorithm 8: ISEXECUTIONGOOD?

Data: Accesses the set of activated monitor conditions $\mathcal{M}_{active}$

Result: A set of conflicts ν, or the empty set $\emptyset$ if no relevant disturbances are detected.

```
1  begin
2      ν ⟵ ∅
3      foreach m = ⟨e_s, e_f, p, r⟩ ∈ M_active do
4          if NOT(ISTRUE?(p)) then
5              Add Conflict({r's constraint, execution trace, ¬p}) to ν
6          end
7      end
8      return ν
9  end
```

predicate p and consuming action a_c. By the definition of an activated causal link, exactly one causal link from each $\mathcal{L}$ will be activated (the one with the latest producer finish time). Therefore, there will be an activated causal link corresponding to each p of each a_c in the plan. □

We show that all of the activated causal links being valid throughout their activation windows is a sufficient condition to ensure that the action's conditions are met by its start time.

Lemma 2.4.10. *If the activated causal link for every condition of an action is valid under $\mathcal{W}$ throughout its entire activation window, the action's conditions will be met by its start time under $\mathcal{W}$.*

Proof. Let $a_c = \langle \boldsymbol{\alpha_c}, e_{s_c}, e_{f_c} \rangle$ be some action in $\mathcal{A}$, with conditions $p_1, p_2, ...p_N$. Further let the causal link candidate sets for each of these conditions be $\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_N$, and the activated causal link in each of these sets be $l_1, l_2, ..., l_N$. Each of these l_i are deactivated at time $t_{e_{s_c}}$, the time at which the conditions must hold true for a_c to be executable. Since we assume that each l_i is valid under $\mathcal{W}$ throughout this window (up to and including $t_{e_{s_c}}$), we can say that each condition $p_i \in \mathcal{W}(t_{e_{s_c}})$. Therefore, the action has all of it's conditions met by its start time under $\mathcal{W}$. □

We are now prepared to prove the two most important theorems of this chapter - namely that the execution monitor is correct. The following two proofs demonstrate that 1.) if all active causal links have been valid, then the execution trace is healthy, and 2.) if some causal link is violated, then the execution trace is not healthy.

Theorem 2.4.11. *During execution if we observe that all activated causal links have been valid in their activation windows for time $t \leq t_{now}$, then the execution trace is healthy at time t_{now}.*

Proof. Proof by construction. Intuitively, our approach to proving this claim is to construct an extension T_{full} to the trace T_{trace}, and show that all activated causal links for this extension will be valid throughout their entire activation windows. We consider case-by-case activated causal links from the past, present, and future with

respect to t_{now}. Since there is an activated causal link for every condition of every action in the plan (Lemma 2.4.9) that is proved to be valid throughout its entire activation window, all conditions will be met and our constructed schedule will be executable. This implies that the current trace is healthy.

We begin by constructing a consistent full schedule extension T_{full} to the current execution trace T_{trace}. For each event in the plan unassigned by T_{trace}, select any value within that event's all pairs shortest path temporal window (as could be computed by Floyd Warshall). As proven in ([5]), any extension constructed in this way from a partial assignment will be temporally consistent, so we have constructed a full-schedule extension T_{full} that satisfies all of the temporal constraints of the plan.

Next, we partition all activated causal links into three sets with respect to t_{now} and our schedule: those occurring in the past (activated and deactivated in $t \leq t_{now}$), those active at present (activated during $t \leq t_{now}$ and deactivated $t > t_{now}$), and those occurring in the future (activated and deactivated in $t > t_{now}$). We show that all of these activated causal links will be valid under $\mathcal{W}_{pred}$ for their entire activation windows.

1. **Past.** Each activated causal link in this set was activated and deactivated in $t \leq t_{now}$ according to T_{full}. We have observed in $\mathcal{W}_{obs}$ that all of these activated causal links have been valid for $t \leq t_{now}$, which entails their entire activation windows. Since $\mathcal{W}_{pred}(t) = \mathcal{W}_{obs}(t)$ for all $t \leq t_{now}$, we can say that all past activated causal links have also been valid under $\mathcal{W}_{pred}$ throughout their entire activation windows.

2. **Present.** These activated causal links have been valid for all $t \leq t_{now}$ under $\mathcal{W}_{obs}$. By an argument similar to the above past case, each causal link $l = a_p \xrightarrow{p} a_c$ has also been valid under $\mathcal{W}_{pred}(t)$ for $t \leq t_{now}$, which accounts for a portion of of its activation window. We need to show that l will also remain valid under $\mathcal{W}_{pred}$ for the other portion of its activation window, which resides in $t > t_{now}$. We begin by noting that all activated causal links are valid under $\mathcal{W}_{pred}$ at time $t = t_{now}$, and for all $t > t_{now}$ recall that $\mathcal{W}_{pred}(t) = \mathcal{W}_{ideal}(t)$. Therefore under

$\mathcal{W}_{pred}$, all predicates true at time t_{now} will continue to hold true indefinitely into the future until modified. Since l is an activated candidate causal link, its producer a_p has the latest possible finish time of all causal link candidates in its causal link candidate set. Additionally noting that by the definition of a causal link no other action may threaten to negate p during its activation window, there can therefore be no other action a' that has a finish time after a_p and before a_c's start time that affects p. Thus, p will continue to hold until a_c's start time, which marks its deactivation time. Therefore, l is valid under $\mathcal{W}_{pred}$ throughout its entire activation window.

3. **Future.** These activated causal links will be activated and deactivated in $t > t_{now}$. We need to show each future activated causal link $l = a_p \xrightarrow{p} a_c$ will be valid under $\mathcal{W}_{pred}$ throughout its entire activation window. By a similar argument to above present case, we know that a_p is the last possible action that can affect p before a_c's start time. Since $\mathcal{W}_{pred}(t) = \mathcal{W}_{ideal}(t)$ for all $t > t_{now}$, p will therefore remain true under $\mathcal{W}_{pred}$ throughout l's activation window. Therefore, l is valid under $\mathcal{W}_{pred}$ throughout its entire activation window.

We have thus shown that all activated causal links - past, present, and future - will be valid under $\mathcal{W}_{pred}$ during their entire activation windows. By Lemma 2.4.9, there is an activated causal link for every condition of every action in the plan. These activated causal links will be valid throughout their entire activation window by what we have proven above. Applying Lemma 2.4.10, we have sufficient conditions to show that every condition of every action in the plan will be met under $\mathcal{W}_{pred}$. Therefore, our schedule T_{full} is executable under $\mathcal{W}_{pred}$, and so the execution trace T_{trace} is healthy. □

Theorem 2.4.12. *During execution if an activated causal link is violated, then the execution trace is not healthy.*

Proof. Let the execution trace be T_{trace}, the current time be t_{now}, the causal link be $l = a_p \xrightarrow{p} a_c$, and the causal link candidates set corresponding to p and a_c from which

l originated be $\mathcal{L}$. For l to be violated at t_{now}, $\neg p \in \mathcal{W}_{pred}(t_{now})$. Intuitively, our approach is to prove that no other action that produces p can finish after a_p finishes and before a_c starts, and hence $\neg p$ will continue to hold in all possible schedules when a_c (which requires p) should have been executed.

By Theorem 2.4.8, the set $\mathcal{L}$ is complete - it contains all causal link candidates for a_c and p. Therefore, there can be no other action a' such that $a_p \prec a' \prec a_c$ that produces p as an effect. Additionally, for l to have become an activated causal link, a_p must have been the last producer candidate of the causal links in $\mathcal{L}$ to finish. There is therefore no action a' that asserts p and is schedulable to finish after a_p's finish event and a_c's start event that can re-assert p.

We wish to check whether T_{trace} is healthy, or namely if there exists an extension T_{full} to T_{trace} that is executable under $\mathcal{W}_{pred}$. Since for $t > t_{now}$ we have defined $\mathcal{W}_{pred}$ to behave like $\mathcal{W}_{ideal}$, $\neg p$ will continue to hold since no other action is schedulable to assert p before a_c. The condition p of a_c can therefore not be met in any schedule, so the execution trace is not healthy. □

Based on the above theorems, we now have a satisfactory condition to describe relevant disturbances. Since the validity of all activated causal links constitutes a healthy execution trace yet a violation of any activated causal link yields an unhealthy trace, we can say that any disturbance in the world that violates the predicate of an activated causal link is a relevant disturbance. Any other disturbance which does not violate a causal link is an irrelevant disturbance.

2.5 Algorithmic Performance Analysis

In the previous sections, we presented algorithms for the Pike plan executive and execution monitor, and proved various properties and guarantees about these algorithms. In this section, we will analyze the computational complexity of these algorithms.

It is very important that we consider the context with which these algorithms are used. In a robotic system, a generative planning algorithm will often be used to generate the plan $\mathcal{P}$ that Pike is given as input. Once generated, Pike will then

proceed to execute that plan while simultaneously monitoring its progress. Generating these plans is however a computationally intensive task that should not be neglected. Planning has been proven to be NP-hard, and as we will show shortly, all of the algorithms we have presented thus far relating to the plan executive and execution monitor have polynomial computational complexity. Therefore, in many practical situations, planning will often be "computational bottleneck," not the algorithms presented in this thesis.

2.5.1 Offline Algorithm Complexity

In this section, we analyze the complexity of the offline algorithms - specifically, PreprocessPlan, GetCausalLinkCandidates, and UpdateProducerCandidates.

Suppose we are given a plan $\mathcal{P} = \langle \mathcal{E}, \mathcal{C}, \mathcal{A} \rangle$ and. Let $|\mathcal{A}|$ denote the number of actions in the plan. We begin with the most deeply nested function, UpdateProducerCandidates, and will work our way upwards. In this function, the set of producer candidate actions ξ, may contain at most $|\mathcal{A}|$ elements ($|\xi| = O(|\mathcal{A}|)$). The worst-case complexity is therefore $O(|\mathcal{A}|)$. We argue however that for most reasonable plans constructed deliberately by a generative planner, ξ will in fact be small. In this case average case, the complexity of UpdateProducerCandidates will be closer to $O(1)$. Since we cannot guarantee the size of ξ however, we will simply leave this value unevaluated and say that the complexity of UpdateProducerCandidates is $O(|\xi|)$.

Next, we consider GetCausalLinkCandidates. We assume that primitives such as checking the existence of a predicate in the effects of a PDDL action take constant $O(1)$ time. This function iterates over each action in $\mathcal{A}$. Then, the complexity of GetCausalLinkCandidates is thus dominated by the first pair of loops and the inner most call to UpdateProducerCandidates, and will operate in $O(|\mathcal{A}||\xi|)$. It's also important to note that the size of the candidate causal link set returned will be $|\xi|$.

We now consider ConstructActionGraph. The first steps are the most com-

putationally intensive - the Floyd Warshall algorithm is cubic in graph size. This dominates the cost of the algorithm, which has complexity $O(|\mathcal{A}|^3)$.

Next, we consider the top-level method PreprocessPlan. The first step constructs an action graph, which has complexity $O(|\mathcal{A}|^3)$. This function iterates over all actions $a \in \mathcal{A}$, and over all conditions p that are preconditions or maintenance conditions of a. Suppose that in the PDDL domain being planned over, each action has at most c conditions (c will vary in different domains, but will generally be a small constant). The outer two layers thus iterate $c|\mathcal{A}|$ times, and the inner-most loop iterates once for every element in $\mathcal{L}$, which has size $|\xi|$. Therefore, the complexity of PreprocessPlan is $O(|\mathcal{A}|^3 + c|\mathcal{A}|(|\mathcal{A}||\xi| + |\xi|))$. Since $|\xi| = O(|\mathcal{A}|)$, the complexity of the algorithm as a whole can be safety approximated as $O(|\mathcal{A}|^3)$ recalling that c is a constant.

As noted above, it is likely in many realistic applications that this polynomial complexity in problem size will be overshadowed by the NP-hard planning problem.

2.6 Experimental Validation

In addition to the theoretical analysis above, we have also performed experimental analysis of the Pike plan executive. We present our results in this section.

2.6.1 Experimental Setup

We wish to measure various properties of the execution monitor, such as its latency in detecting relevant disturbances during execution and the time required for the preprocessing stage.

In order to make such measurements, we developed a testing and benchmarking platform capable of automatically generating random PDDL problems and corresponding temporal plans. This platform is also capable of simulating the plant for arbitrary PDDL domains, thereby allowing Pike to control a simulated environment in which disturbances can arbitrarily be injected. The plant responds to PDDL actions dispatched by the executive and updates the world state accordingly, incorporating a

random disturbance model that may randomly at any point remove some predicate from the world state. The plant additionally broadcasts the world state at a constant, high-frequency interval for processing by the execution monitor. By measuring the times at which the disturbance is injected into the system and the time at which the failure is signaled, we may benchmark the execution monitor's latency.

Random plans are generated by generating sequences of random executable actions which when pieced together form a plan with a well-defined goal state. In this way, we are able to easily generate PDDL problems and plans with a controlled number of actions for PDDL domains.

We characterized the latency of the execution monitor using our benchmarking platform and a temporal variation of the BlocksWorld domain. Specifically, we generated random plans with 5 actions each, and used Pike to dispatch these plans online. At some randomized time during the middle of executing each trial, a disturbance was injected in the form of removing a random PDDL predicate from the world state. If the execution monitor signaled failure (the random disturbance was relevant), the latency is measured. This experiment was performed thousands of times.

Please see Figure 2-14 for a latency histogram of the execution monitor. Please note that typical latencies are around 20-60ms, which is on the same order of magnitude as some implementation-related socket wait calls). Additionally, we must note that the execution monitor checks received predicates at a rate of 100Hz. These results were generated using the BlocksWorld domain, with plans of 5 actions long over thousands of trials. We note that the latency time is generally very small, and empirically quite tolerable for reactive fault detection on real robotic systems.

In addition to latency, we also empirically characterized the time required to preprocess a plan and extract sets of candidate causal links. Please see Figure 2-15. This plot illustrates the preprocess time as a function of the number of actions in the plan, which range from 1 action to roughly 150 actions. This plot is roughly cubic as expected, and is dominated by the computation of the all-pairs shortest path used for evaluating the $\prec$ relation for actions in the plan. We suspect that the small "dips"

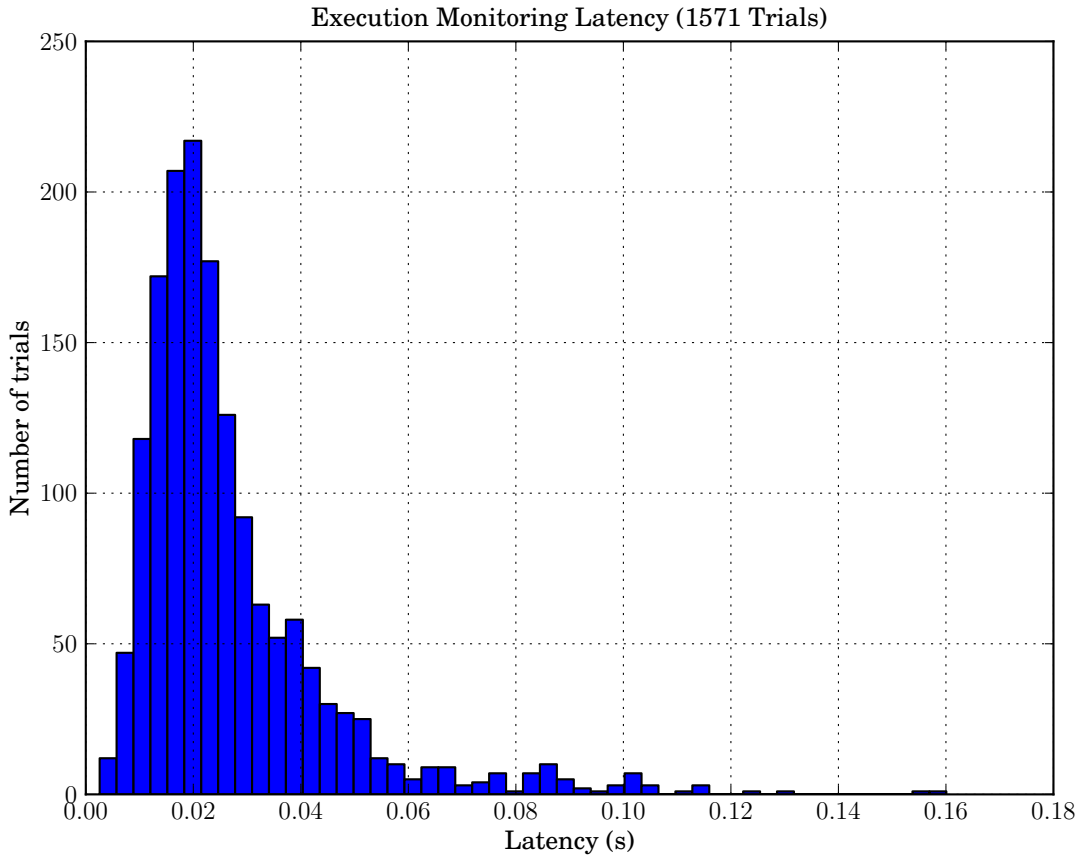

Figure 2-14: The latency, measured in seconds, of the execution monitor. Thousands of PDDL blocks world domains were randomly generated, along with random plans with 5 actions each. Pike was then tasked with executing these random plans. During execution, at some random time point, a disturbance (in the form of removing some predicate from the world state) would be injected, and the time differential from when the disturbance was injected to when the execution monitor signaled failure (if a causal link was violated) was measured.

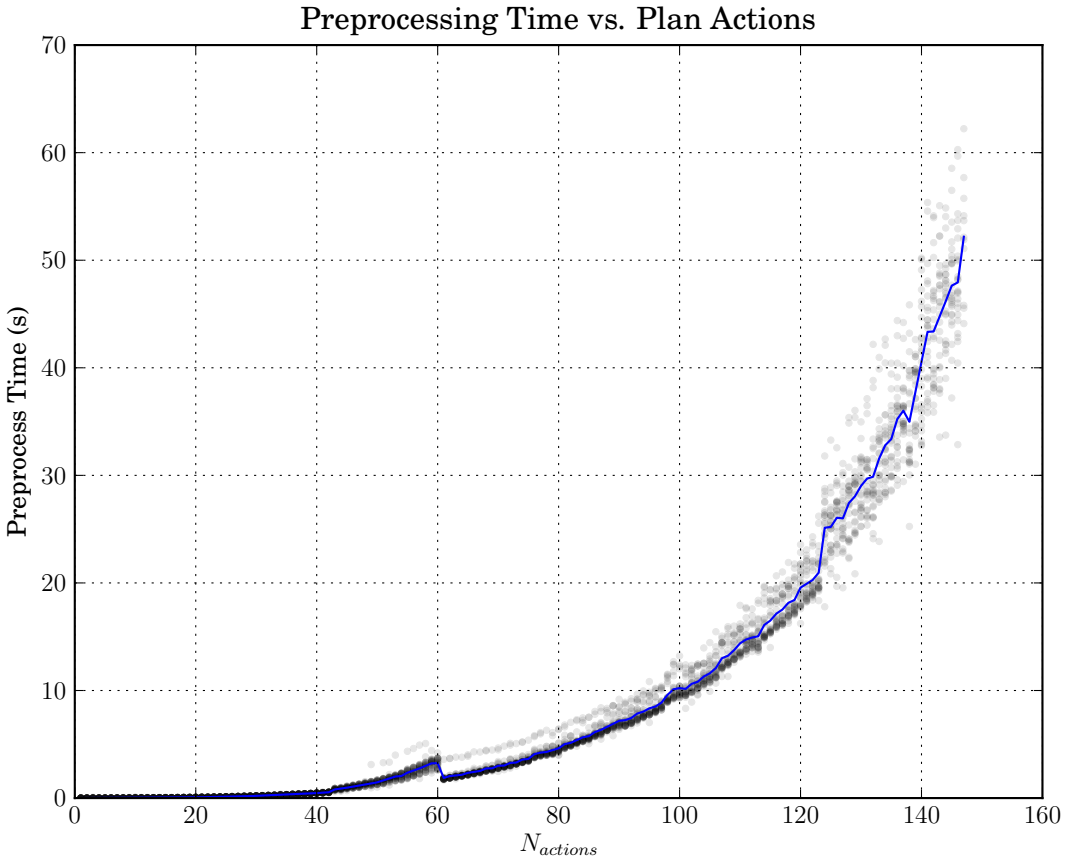

Figure 2-15: The preparation time in seconds as a function of plan size. This plot is approximately cubic, and is dominated by the computation of the Floyd Warshall APSP algorithm. The black dots illustrates the raw preparation times from each trial, and the blue line averages each set of $N_{actions} = k$ for some k over its 25 trials.

in the graph are the result of our LISP environment performing online hash table optimization.

We additionally note that, though the system at present may take up to a minute for large plans of approximately 150 actions, these plans are much larger than typical plans for systems. In these situations, a generative planner would likely take much longer to generate such a plan. For more realistic plans of 20 actions or less, the preparation times were considerably more reactive.

2.7 Chapter Summary

In this section, we introduced the core technical work of this thesis. Namely, we presented Pike, a plan executive capable of dispatching and monitoring temporally flexible plans consisting of actions with preconditions, maintenance conditions, and effects. These algorithms are capable of quickly and efficiently deducing whether or not an execution trace in progress is healthy, or if replanning is needed because the plan will not succeed. Our approach to solving this problem was to extract sets of candidate causal link sets offline before dispatching is done. The extraction of candidate causal links, and the non-existence of a unique set of these links, is a core novel contribution of this thesis.

Once sets of candidate causal links have been extracted, they are monitored online as the plan is dispatched. This allows Pike to immediately detect any relevant disturbances that can jeopardize the plan.

In the subsequent chapters of this thesis, we build upon the Pike plan executive and use it to construct further robust systems. In the next chapter, we combine Pike with a temporal generative planner called TBurton in order to create the TBurton Executive, a system capable of robustly meeting a users goals through replanning when the execution monitor signals failure. In the following chapter, we make use of this idea of a generative planner combined with the Pike plan executive to demonstrate an innovative application implemented in hardware: a robust, voice-commanded robot capable of stacking blocks.

Chapter 3

The TBurton Executive

Broadly speaking, a planner is responsible for computing a sequence of actions to be carried out by a robotic agent that will (in theory) take the world from some initial state to a given desired end state, given an appropriate model for how the world behaves. Once such a sequence of actions is generated, a separate module is then responsible for actually executing the plan and ensuring that it is carried out properly. The union of these two pieces, namely the planner and the execution component, is collectively called the executive.

In this chapter, we introduce the TBurton executive, a system capable of generating plans and subsequently executing them robustly with Pike in order to meet a user's goals. Goals are specified at a high level that is intuitive to the user, and execution is guaranteed to be robust to a number of possible disturbances.

We begin this chapter by providing an example to motivate the TBurton exectutive. We then boil down the salient features of this example into a set of key requirements that the TBurton executive must meet. Next, we introduce our system architecture to meet this requirements, defining the input/output relations of each submodule. This architecture makes use of the Pike plan executive with execution monitoring developed in the previous chapter.

Figure 3-1: At left, an illustration of a mass-production automotive facility. The robots execute the same task repeatedely with no variation, thanks to a precisely-designed environment where nearly all variability is engineered away. At right shows a typical modern aircraft manufacturing facility. Due to the different nature of the product being produced, mass-manufacturing techniques and a highly-engineered environment aren't easily implementable.

3.1 Motivation

Today's mass-production factories draw a clear separation between the tasks that robots perform and those that are carried out by people. The robots are often used for monotonous tasks that are constantly repeated with little or no variation, whereas people are typically used for the more advanced tasks in which problems may more easily arise. For example, in a typical car manufacturing plant, robots may perform welding or painting operations by repeating pre-programmed instructions over and over for each car that comes down the assembly line. Since each car comes down arrives at precisely the same location with respect to the robot, no changes or alterations to the plans are required; the robots may playback their preprogrammed command sequence. People on the other hand are used for more complicated assembly tasks that require fine dexterity and inspection abilities - namely those tasks where there is variability and decisions may need to be made. However, in certain manufacturing settings, a precisely engineered environment is not always feasible. As such, it would be very useful to endow our robots with a greater sense of autonomy and ability so that they will still be able to contribute to the manufacturing process. Figure 3-1 illustrates this.

To motivate the usefulness of the TBurton executive, we will present here a fictitious story illustrating our vision for the system set in the context of robotic manufacturing.

Consider an airplane manufacturing facility of the future in which teams of autonomous robots work hand-in-hand with teams of humans as equals. These robots are capable of reasoning about their environment, and have a rich model of the world that enables them to make informed decisions.

During the process of assembling a wing for one of the company's airplanes, a human operator may ask a robot, "Robot, please finish assembling the winglet and bring it to pallet 4 when you're finished. We need this done in 20 minutes so that we stay on our tight schedule."

Once given a set of commands, the robot is able to autonomously go and complete it's task. Using it's world model and the desired state goals (finished wing assembly in pallet 4) and temporal constraints (complete in 10 minutes), the robot generates a plan of action for how to proceed. Such a plan may include the following steps:

1. Locate the winglet in the factory and go there [0 - 1 minutes]
2. Finish tightening the bolts and make the final welds to complete the winglet [3 - 6 minutes]
3. Pick up the winglet [30 seconds]
4. Transport it to pallet 4 [3 - 6 minutes]
5. Place the winglet down [30 seconds]

These actions, under nominal circumstances, will allow the robot to complete its task successfully. However, a key observation is that there are a number of opportunities for things to go wrong with respect to this plan. For example, the robot may notice that one of the bolts is damaged, or the robot may find that there is already some other object sitting on pallet 4 and hence the winglet cannot be put down there. In each of these situations, some assumption made by the robot when generating it's

initial plan has been violated. A solution is to replan given the current knowledge of the world. Taking the example where there is already another subassembly on pallet 4, the robot may generate a new plan to put the winglet on pallet 5 if it is free. This, however, may cause the robot to violate it's temporal constraints, as it may need to drive considerably farther to get to pallet 5. In this case, there may be no feasible plan that will satisfy all of the user's desired constraints - a negotiation process is therefore required to find a desired way to relax certain constraints in a satisfactory manner.

3.1.1 Key Requirements of the TBurton Executive

In this section, we will boil-down the salient features that we desire for the TBurton executive from the above scenario. These are the most important aspects of the TBurton executive, and our architecture is central to achieving these requirements.

1. **Goal specification in a high-level language.** The factory worker was able to command the robot intuitively through language at a high level by specifying goals to be accomplished, not details on how to accomplish them.

2. **Execution monitoring to detect relevant problems during execution.** Executing plans open-loop without sensor feedback is very brittle due to disturbances and uncertainty in the world. Closed-loop feedback in the form of execution monitoring is a requisite to achieving robust task execution in the face of these difficulties.

3. **Recovery through replanning.** Once a robot detects potential problem in the world, it must chart out a new course of action that will achieve the desired goals within the desired temporal constraints.

4. **Generate least-commitment plans that maximize execution flexibility.** The plans generated by the robot should contain built-in flexibility in the form of set-bounded temporal constraints.

3.1.2 TBurton Executive Problem Statement

In this section, we intuitively and then formally define the problem statement for the TBurton executive. Please note that we will make use of the formalisms developed in the previous chapter of this thesis.

From a high level, the job of the TBurton executive is to robustly meet a users goals in the face of unexpected disturbances. It must take in the user's goal in a high level format that specifies what the user wants, not how to accomplish it. For example, we wish to command the robot with goals such as "Build a red tower of blocks in no more than 2 minutes. Once you've finished, wait 3 minutes and build a blue tower of blocks next to it." Notice that this goal specification actually specifies multiple different subgoals (red tower and blue tower) in addition to allowing temporal constraints to be specified.

The TBurton executive must additionally be given a set of operators describing possible actions in the world. These operators will form the basis for the actions in the plan that TBurton generates to achieve the goals. These operators may additionally have temporal constraints in the form of variable durations.

Finally, the TBurton executive must have access to world state estimates in order to deduce the state of the world. This is necessary for detecting when a disturbance jeopardizes the plan.

We now present an intuitive definition of the TBurton Executive. The TBurton Executive takes in the following inputs:

- A high-level goal specification consisting of state requirements and temporal constraints
- A set of operators describing the possible actions the TBurton executive may execute
- Estimates of the world state over time

It generates the following outputs:

- A dispatch of actions whose preconditions, maintenance conditions, and effects are met such that the goal conditions are satisifed, or FAILURE if at any point it is determined that no such sequence of actions can exist.

In the following sections, we will introduce formalisms for the above in greater detail. We will then proceed to reformulate the TBurton executive problem rigorously using these formalisms.

3.1.3 Qualitative State Plans (QSP's)

In this section, we will describe the TBurton executive's goal input form, namely the Qualitative State Plan (QSP), in greater detail.

Unlike the goal form used in classical planners, which consists just of a conjunction of PDDL predicates, the QSP additionally captures the notion of timed goals. If we want to specify that some subgoal or conjunction of predicates hold true during one time window, and another hold true during a different time window, then we would need to use a goal representation similar in spirit to QSP's. Intuitively, QSP's are very similar in formal structure to the temporal plan we described in the first chapter of this thesis. However, instead of containing PDDL actions that label pairs of edges, we instead use conjunctions of PDDL predicates representing goal conditions that must satisified during that time window.

The QSP was used extensively in the goal represention for work in generating reactive execution policies for a bipedal walking robots ([16]). The word "Qualitative" refers to the hybrid-nature of this work, in which different controllers were used to control the bipedal model in different regions of state space, where each different region was called a qualitative state (for example, toe off, toe strike, etc). We now use qualitative for a similar purpose, in that we specify some properties about the world we desire to be true.

Formally, we define a QSP as follows:

Definition 3.1.1 (Qualitative State Plan)**.** We define a Qualitative State Plan (QSP) as a tuple $\langle \mathcal{E}, \mathcal{C}, \mathcal{S} \rangle$, where:

- $\mathcal{E}$ is a set of events. Each event $e \in \mathcal{E}$ is associated with a specific point in time $t_e \in \mathbb{R}$. Additionally, there is some distinguished event $e_{start} \in \mathcal{E}$ that represents the first event and for which $t_{e_{start}} = 0$.

- $\mathcal{C}$ is a set of simple temporal constraints over the time points in $\mathcal{E}$.

- $\mathcal{S}$ is a set of state requirements. Each $s \in \mathcal{S}$ is a tuple $\langle \boldsymbol{\sigma}, e_{start}, e_{end} \rangle$. $\boldsymbol{\sigma}$ is a conjunction of PDDL predicates specifying desired goals that must hold TRUE for all t in the interval $t_{e_{start}} \leq t \leq t_{e_{end}}$.

As can be seen above, the QSP is defined very similarly to the temporal plan, with the exception that conjunctions of PDDL predicates are used to label certain edges instead of executable PDDL actions as in a temporal plan.

3.1.4 Satisfaction of a QSP

We now define one final concept in order to introduce how a QSP can be evaluated. Intuitively, we wish to determine whether all of the goal conjunctions in a QSP hold at their proper times. This necessitates some way to evaluate the state of the world at particular times, and so we hence make use of the world state function formalisms developed in the previous chapter. We define what it means for a schedule of a plan to *satisfy* $\mathcal{Q}$ *under* $\mathcal{W}$:

Definition 3.1.2 (Schedule Satisfying a QSP)**.** Given a schedule T_{full} for a plan $\mathcal{P}$, and a QSP $\mathcal{Q} = \langle \mathcal{E}, \mathcal{C}, \mathcal{S} \rangle$, we say that the schedule T_{full} satisfies $\mathcal{Q}$ under $\mathcal{W}$ if for each $s = \langle \boldsymbol{\sigma}, e_{start}, e_{end} \rangle \in \mathcal{S}$, $\boldsymbol{\sigma} \subseteq \mathcal{W}(t)$ for all t in the range $t_{e_{start}} \leq t \leq t_{e_{end}}$ (Please note that the actions of $\mathcal{P}$ may implicitly be required in the evaluation of $\mathcal{W}(t)$).

This concept is analogous to that of a schedule being executable for a temporal plan $\mathcal{P}$. We now define a concept for QSP's similar in spirit to the healthiness of an execution trace. Namely, we wish to query whether it is possible for the current execution trace to possibly result in a full-schedule extension where $\mathcal{Q}$ is satisfied. We therefore define what it means for an execution trace to be *on track to satisfy* a QSP:

Definition 3.1.3 (Satisfiable Execution Trace)**.** Given an execution trace T_{trace} for a plan $\mathcal{P}$ and a QSP $\mathcal{Q} = \langle \mathcal{E}, \mathcal{C}, \mathcal{S} \rangle$, we say that the execution trace T_{trace} *is on track to satisfying* $\mathcal{Q}$ if there exists a full-schedule extension to T_{trace} that is satisfied under $\mathcal{W}_{pred}(t)$.

3.1.5 Formal Problem Statement

By this point, we have defined all of the formalisms necessary to define the TBurton Executive planning problem. We do so here in order to augment our intuitive description provided earlier with a more rigorous definition now that the requisite formalisms have been defined.

The TBurton Executive takes in the following inputs:

- A Qualitative State Plan $\mathcal{Q}$ describing time-evolved goals to be met
- A set of PDDL operator schema $\{\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3, ...\}$ describing the possible actions the TBurton executive may execute
- Observed world state estimates $\mathcal{W}_{obs}(t)$ valid for all time $t \leq t_{now}$

It generates the following outputs:

- The execution of a plan $\mathcal{P}$ (which may be modified several times throughout the course of execution) such that the finalized schedule resulting at the end of execution satisfies $\mathcal{Q}$ under $\mathcal{W}_{pred}$.
- The TBurton Executive must return immediately should it be determined that there is no $\mathcal{P}$ where the current execution trace could satisfy $\mathcal{Q}$.

3.2 TBurton Executive Algorithms

In this section, we present the core algorithm for the TBurton executive. This algorithm makes use of two subcomponents: The TBurton generative planner, and Pike.

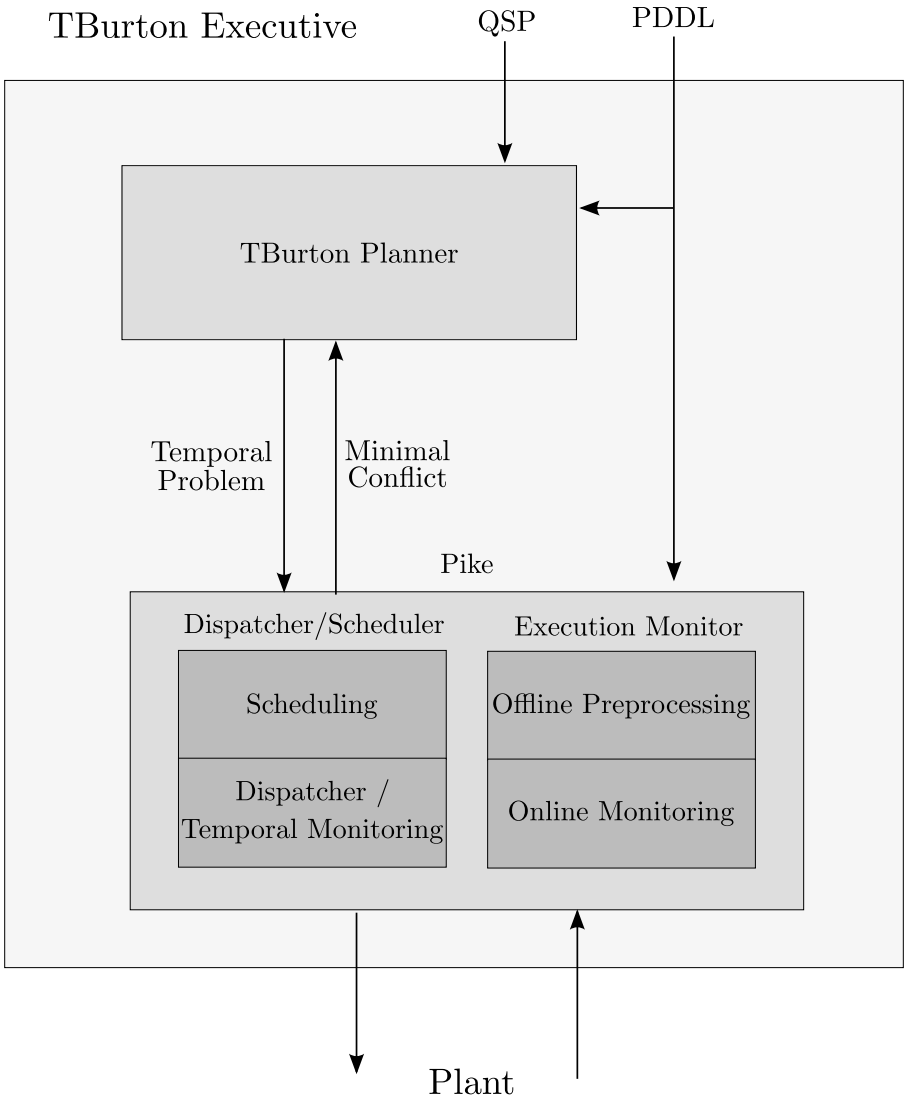

Figure 3-2: The architecture of the TBurton executive. The key components are the TBurton Planner, responsible for generating temporal plans given a QSP (qualitative state plan) input with an action schema specified in PDDL, and Pike, responsible for dispatching the plan and monitoring its progress online. The dispatcher sends command out which are implemented by the plant. State estimates return to the online phase of the execution monitor.

Algorithm 9: TBurtonExecutive

Data: World state observations $\mathcal{W}_{obs}(t)$ to determine the initial state, a goal specification in the form of a qualitative state plan $\mathcal{Q}$, $ops = \{\mathcal{O}_1, \mathcal{O}_2, ...\}$

Result: A mapping $\mathcal{D}$ of events to sets of candidate causal link sets, or Failure if the plan is not executable under $\mathcal{W}_{ideal}$

```
1  begin
2      Conflict ⟵ ∅
3      P ⟵ Null
4      repeat
5          (P, f) ⟵ TBurtonPlan(W_obs(t_now), Q, ops, t_now, P, T_trace, Conflict)
6          if f then
7              ⟨Success, Conflict⟩ = Pike(P, ops)
8          else
9              return Failure
10         end
11     until Success = True
12 end
```

The TBurton planner is a temporal, generative planner goal-regression planner. TBurton stands for "Temporal Burton", and is similar in spirit to the Burton planner developed in [30]. The Burton planner takes as its input a concurrent transition system, which specifies a set of variables (partitioned into state variables and control variables), domains for those variables, and transitions described in propositional logic that map states to next states. Burton operates by compiling this transition system into a representation that allows very efficient online execution without an explicit mechanism to detect threats.

For the purposes of this discussion, we model it as a PDDL planner in that it is capable of taking in a QSP $\mathcal{Q}$, initial world state, and set of PDDL operator schema as input and return a temporal plan $\mathcal{P}$, all of the consistent schedules of which satisfy $\mathcal{Q}$ under $\mathcal{W}_{ideal}$. While the native input representation to TBurton is not actually in fact PDDL (it is a timed concurrent constraint automata model), the TBurton Planner is capable of converting between these two representations.

We assume that the function TBurtonPlan$(\mathcal{W}_0, \mathcal{Q}, ops, t_{now}, \mathcal{P}_0, T_{trace}, Conflict)$ returns a tuple $\langle \mathcal{P}, f \rangle$ where f stands for "found" and is a binary variable indicating

whether or not a plan $\mathcal{P}$ was found that satisfies $\mathcal{Q}$ under $\mathcal{W}_{ideal}$. The input plan $\mathcal{P}_0$ and corresponding executiont race T_{trace} is some initial plan, which is modified by inserting new actions at proper times. The TBurtonPlan function also takes in the current world state, a desired QSP, and a set of PDDL operator schema. The $Conflict$ argument allows a conflict to be provided that may speed up the search process. This is the conflict that is returned by the Pike plan executive.

An architecture diagram for the TBurton executive is shown in Figure 3-2, and psuedo code for the algorithm is shown in Algorithm 9. Intuitively, the algorithm proceeds by continually building up a temporal plan $\mathcal{P}$ using the TBurton planner. The plan is then executed using Pike. If Pike returns an unsuccessful result because of a disturbance that occurs, the TBurton Planner is continually re-called (thereby adding more actions to $\mathcal{P}$ and advancing in time) until either Pike finally succeeds at executing the plan (and hence the users goals have been met), or there is no such possible and Failure is returned.

The TBurton Executive is very "persistent" at attempting to reach its goal. Should the execution trace for a current plan be deamed unhealthy by Pike, the TBurton Executive responds by generating a new plan that should in theory be executable. This allows the robot system to chart a new course of action and overcome disturbances.

3.3 Chapter Summary

In this chapter, we introduced the TBurton Executive, a system that makes use of the TBurton temporal generative planner and the Pike plan executive to robustly meet a user's goals that are specified in a high-level form. The TBurton executive is a continuous planning and execution system, and reacts to disturbances in the world that would otherwise preclude successful execution of the current plan by generated a new, augmented plan that will meet the users goals.

Chapter 4

Innovative Application: Voice-Interactive Robotic Manufacturing

While there have been countless discussions over the years as to what constitutes true "intelligence" in artificially intelligent robotic systems, many (including the author) argue that high-level interaction with the robotic agent is a key component. A robotic system that is able to accept user-specified commands at a high level, reason over them sufficiently to execute them even in the face of disturbances, and subsequently explain any rationale behind choices made or problems encountered seems to be, at least to the author, a very good indicator of an artificially-intelligent system. In this chapter, we introduce a robot that aims to meet these goals in a limited context. Specifically, we design a robot - tested in hardware and in simluation - that is capable of robustly meeting user specified goals in the ever-present face of disturbances, while simultaneously verbalizing any problems that arise at an intuitive, high level to the user.

Our system is set within the context of robotic manufacturing. We implement an executive similar in spirit to the TBurton executive (but using the FF planner instead) as a key subcomponent of a system demonstrating robust recovery from failure as well as voice-commanded robotics. We begin by re-iterating our vision for

the future of robotic manufacturing. Then, we continue on to describe the capabilities and implementation of our robotic system.

4.1 Robotic Manufacturing: Our Vision

As discussed in an earlier chapter of this thesis, current robots employed in factories are often used to perform repetitive tasks with little to no variability. While capable of high efficiency in well-controlled settings, such robots encompass a minimal ability to respond to failures that may arise or to adapt autonomously to new situations. Additionally, humans are not permitted near such robots due to their lack of sufficient sensors for safe operation.

We envision a future in which humans and robots will work alongside one another in a factory setting, working as a team to accomplish their fabrication-oriented goals. Of the many requirements for such robots, we focus on the following:

- Fluent and natural communication with the robot using high-level language
- Ability of the robot to autonomously adapt to new situations and disturbances

We have implemented and tested a simplified version of such a robotic manufacturing task that meets these goals, and will describe it in the next section.

4.2 System Capabilities

We will now describe the capabilities of our robotic manufacturing agent designed to work alongside humans. A picture of our hardware testbed is shown in Figure 4-1.

The robotic manufacturing scenario consists of a WAM (Whole Arm Manipulator) produced by Barrett Technologies complete with a hand, a number of brightly colored boxes marked with fiducial tags, a rolling cart, and a number of inexpensive webcams (not shown). The goal of this scenario is for the robot to autonomously create assemblies of blocks which are stacked on top of the cart despite any number of disturbances that may occur.

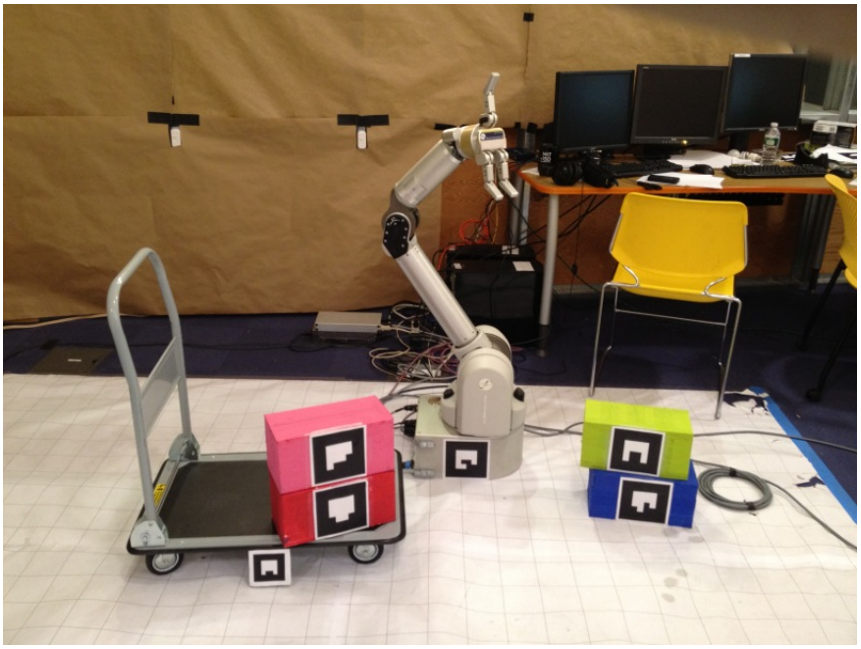

Figure 4-1: Our implementation of a robotic manufacturing agent. The Barrett WAM (Whole Arm Manipulator) is capable of manipulating colored blocks, which are each marked with a fiducial for sensing purposes. It is capable of robustly creating assembling stacks of blocks on the cart.

We argue that this situation encompasses many of our key goals for intelligent factory robots. Although robots working in a real aerospace manufacturing facility would likely be tasked with more complex tasks in terms of manipulation and grasping (for example, a real manufacturing robot may be welding, screwing in bolts, or handling assemblies with more complicated geometries), we feel that our simplified block-stacking robot strikes a good balance between the practicality of implementing an academic demo and demonstrating our key research algorithms, which are the focus. We aim to focus on the planning, monitoring, and verbal interaction aspects of this robot rather than it's motion planning and grasping details, which are currently out of the scope of this project.

When the scenario begins, a human co-worker approaches the robot and identifies him or herself, saying "This is Steve.". The robot then politely asks the person, "Hello, Steve. What can I do for you today?" To this, the person may give any number of possible commands, such as "Make assembly A", "Unload the cart", etc.

Upon making such a request, the robot will immediately proceed to accomplish it's goal. It autonomously generates a plan for how to achieve the goal using the generative planner. Additionally, while the plan is executing, the array of webcams continuously monitor the world state by tracking the fiducial locations on the blocks and cart. Should a disturbance occur, such as a block falling or being moved by a person, the execution monitor will detect it and, if necessary, signal for a re-plan. In this way, the robotic agent is robust to failure. Blocks may be moved around by a malicious or a benevolent person. Each time a disturbance relevant to the plan occurs, the robot will automatically discern a new course of action to reach the goal state and execute it.

4.3 System Architecture

In this section, we describe the overall system architecture and the theory of operation of our demo from a high-level. The next section of this thesis will discuss each section in greater detail and include implementation information.

From a high level, the software architecture consists of the executive, an Activity Dispatcher that is responsible for actually executing the PDDL commands dispatched by Pike, a simulation, a computer vision module, a hybrid state estimation module, and a speech module. The executive is implemented in LISP, and the rest of the system is programmed in Python or C++. The Robot Operating System, or ROS, is used as the glue that binds the modules together. Additionally, the simulator used in our scenario heavily builds upon the OpenRAVE simulator [7].

When the demo begins, the speech module has a short conversation with the person in order to extract his or her goals for the robot. Upon receiving a verbal command such as "Make assembly A", it is forwarded to the executive where various commands correspond to PDDL goal conditions, namely conjunctions of instantiated PDDL predicates. For example, the "Make assembly A" phrase maps to the PDDL goal `(and (on-cart RedBlock Cart) (on MediumPinkBlock RedBlock))`. Other commands, such as "Make assembly B" or "Unload cart" have similarly defined map-

pings. Using these goal conditions and a set of initial conditions (also instantiated PDDL predicates), the executive creates a plan and begins dispatching it on the robot. Upon dispatching a PDDL action, Pike sends the action (via a ROS message) to the Activity Dispatcher, which is responsible for coordinating execution in the simulator and calling any necessary motion-planning libraries. The resulting trajectories are then sent to a lower level, to the WAM hardware controller that communicates via CAN bus to the WAM arm and physically actuates the robot and sends proprioceptive sensory information such as current joint angles back up into the simulator.

The visual tracking system is running at all times, and consists of a number of ROS packages designed to track the black and white coded fiducials placed on the blocks and cart. This data is however in practice extremely noisy, so a series of filtering algorithms are used to extract more meaningful pose data, which is re-routed back to the simulation to update the estimated positions of the blocks. The simulation then sends the block locations and robot kinematics to the state estimation module, which attempts to estimate the PDDL predicates that are believed to be true in the world (this is the level of abstraction at which the executive operates). These state estimates are in turn used by the Execution Monitoring module within Pike.

A key architectural choice is that all of the hardware control and sensing information goes through the simulation environment before being dispatched to the actual hardware. This was a conscious decision, and allows us to purposely turn off the hardware and just use the simulation alone if we so desire. In this case, the simulation environment can simulate faults or random disturbances. Such a design allows the software to be run on any computer, irregardless of whether a robot is attached or not, and also allows for extensive automated testing in simulation.

4.4 System Components

In this section we'll describe each component in greater detail, including implementation details. Figure 4-2 shows the complete implementation diagram.

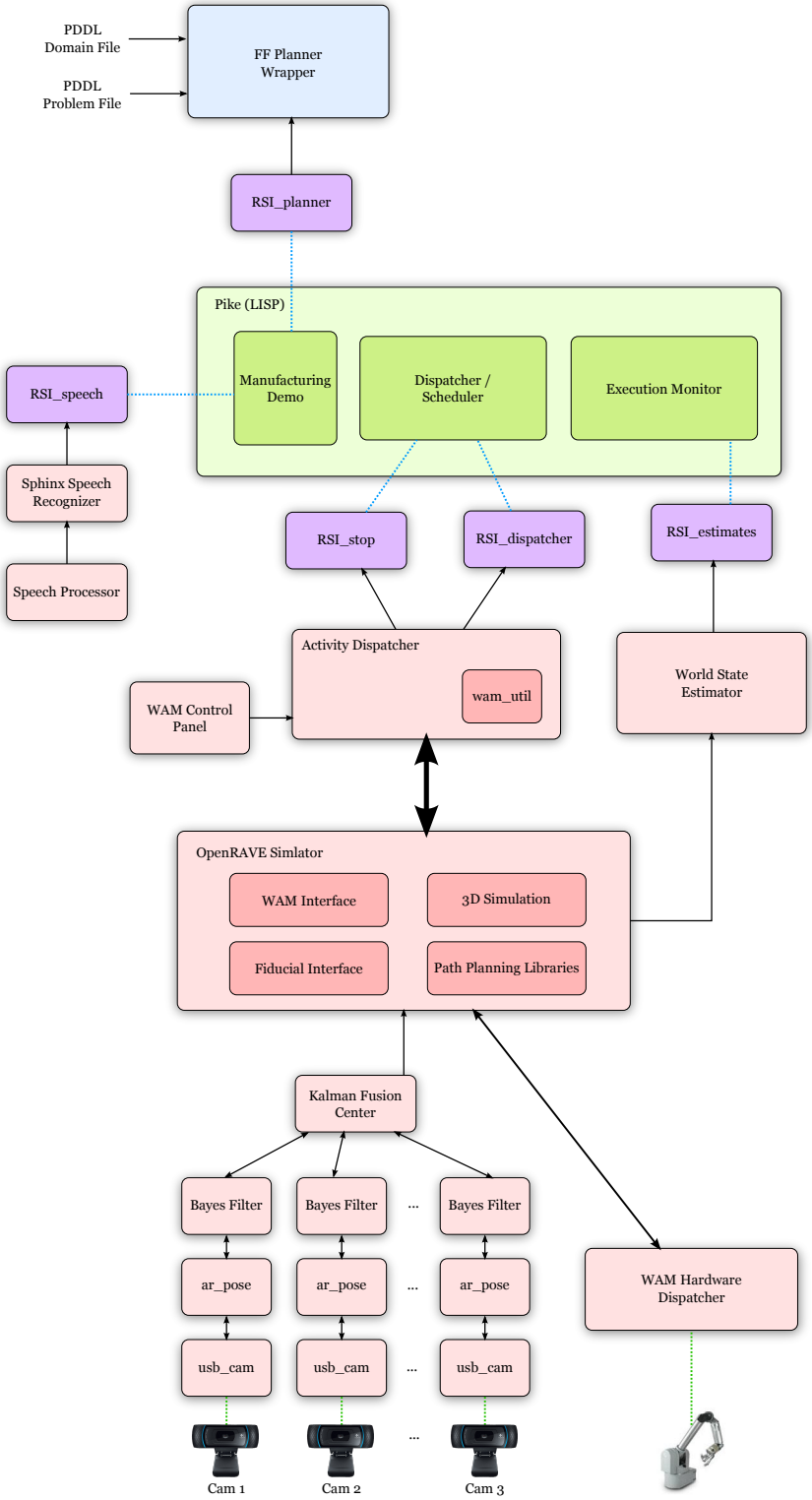

Figure 4-2: The system architecture, as implemented.

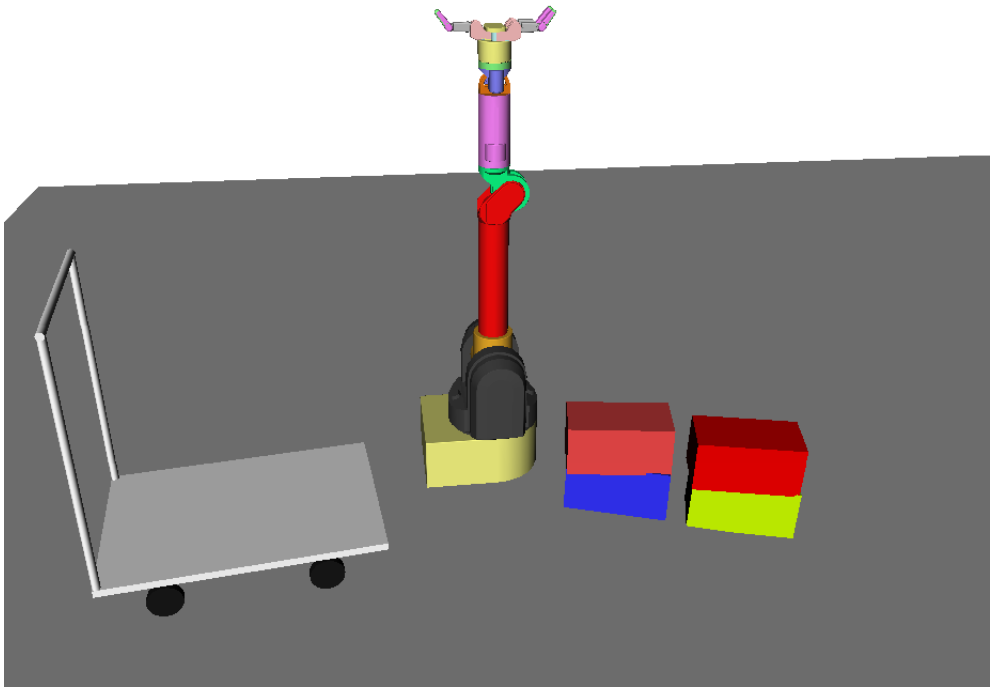

Figure 4-3: Our simulation environment, which builds upon the OpenRAVE environment for it's 3D display as well as motion-planning capabilities.

4.4.1 Simulation Environment

We begin by describing the simulation environment, which aside from the executive, is a central piece of architecture. Our simulation environment, a screenshot of which is shown in Figure 4-3, builds upon the open source OpenRAVE platform [7]. OpenRAVE provides a number of useful tools, such as a 3D simulation and collision detection environment, robot models, a number of motion planning functions such as BiRRT, and convenient Python and C++ language bindings. We use OpenRAVE with it's Python bindings to implement functionality within our testbed. Aside from the Execution Monitoring, implementing this simulator was a key piece of my work in support of this thesis.

As alluded to briefly earlier, the simulation environment is capable of operating in two modes: 1.) Pure simulation in mode, in which no hardware or external sensors are attached, and 2.) Hardware-Mirroring mode, in which the simulator mirrors the

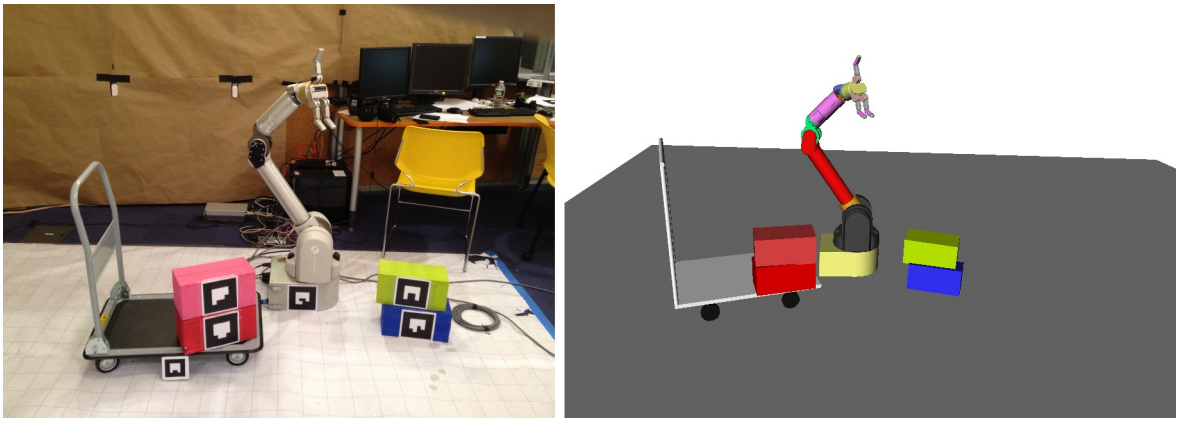

Figure 4-4: Simulation Mirroring of the Hardware. Using the camera array and proprioceptive sensors of the WAM arm, a 3D model can be constructed that accurately reflects the real environment, mirroring changes in real time. Note the similar pose in the testbed and simulation above.

hardware.

Pure simulation mode is useful for performing extensive automated testing where a person does not need to be present. It is also useful in situations where a robot is unavailable, hardware is being repaired, or it is desired to run the software away from our lab. In this mode, the robot is assumed to have perfect joint position control, and the location of all of the blocks are assumed to be perfectly observable with no measurement noise. It is easily possible to extend the simulation to simulate measurement noise, however. It is additionally possible to easily extend the simulator to insert random disturbances, such as blocks moving on top of each other spontaneously, for automated testing purposes.

In Hardware-mirroring mode, the simulation is connected to the hardware through all of the modules presented below the Simulator in Figure 4-2. Intuitively, the simulator attempts to reproduce the state of the real world, as shown by the similarity of the two in Figure 4-4. Namely, proprioceptive sensory information in the form of joint angles are sent from the WAM arm hardware controller to the simulator to update the kinematics of the arm in simulation (and also the computed gripper pose). Sensory information from the visual tracking module is also sent to the simulator, where it is used to update the poses of all of the blocks and the WAM arm with respect to each

other, thereby providing a realtime updated model of the world from within the Simulator environment. Finally, all motion plans that result from dispatching activities are forwarded to the WAM hardware controller. Thus, in hardware-mirroring mode, the simulator provides a transparent interface to the hardware. Switching between hardware-mirroring and pure simulation modes is as simple as running the Simulator with a different command line argument.

The Simulator provides a number of key pieces for our scenario, the most obvious of which is a 3D display showing the current world state. This is invaluable for providing debugging information and for understanding the internal model of the world used by the robot.

Additionally, OpenRAVE provides a number of tools that we leveraged in constructing our demo. Kinematic models of the Barrett WAM arm existed, as did tools for easily creating blocks, carts, and other objects to be visualized in the world. We made extensive use of these. OpenRAVE also provides excellent motion planning capabilities and the ability to compile an efficient inverse kinematics database for a robot. Our WAM arm has seven degrees of freedom (not including the fingers or hands), and OpenRAVE was able to provide inverse kinematics and joint angle trajectories via a BiRRT for picking up and manipulating the blocks.

4.4.2 WAM Hardware Testbed

In this section, we will describe the physical hardware in the testbed. We use a seven degree of freedom Barrett WAM (Whole Arm Manipulator) equipped with a 3-fingered Barrett Hand. The WAM arm is quite dextrous and fast, and it is also extremely backdrivable and void of backlash due to its use of a cable-differential drive system instead of the more traditional gearing system used on many other robots. With seven degrees of freedom, the arm is kinematically redundant; there are a number of possible joint angles that yield the same end effector pose. It is possible to command arbitrary cartesian positions and orientations simultaneously around the robot, making it ideal for picking up oddly-placed blocks in our testbed.

The Barrett Hand contains three independently actuatable fingers, two of which

can also spin around symmetrically about the middle. Additionally, the secondary joint on each finger has a slip differential such that when the first joint makes contact with an object and stops, the second joint will then continue moving. This usually results in a firm grip on the object being manipulated.

Both the WAM arm and the Barrett Hand are controlled via an external PC. This computer is connected to the WAM arm via a CAN bus, and to the hand via standard RS232 serial. Our external PC is running Ubuntu 11.04 (required for ROS diamondback) with a patched kernel modified to support Xenomai. This gives the Linux kernel real-time scheduling support, allowing us to use the CAN bus to control the WAM arm. We originally used Barrett's older btdiag library to control the arms, but have since upgraded to their newer C++ libbarrett library. Our code uses this library for the real-time control of the arms. The WAM hardware controller module noted above serves as a waypoint dispatcher to the hardware, and it additionally publishes the WAM's joint angles periodically to the simulator.

4.4.3 Visual Sensing System

The visual sensing system, coupled with the World State Estimator, is what allows the execution to be monitored. It was implemented primarily by Pedro Santana in the MERS group here at CSAIL, and does an exceptional job tracking and filtering the fiducials attached to the colored blocks and cart. We use several standard, inexpensive webcams, whose precise locations need not be calibrated, to accurately track the poses of all fiducials in 3D. We make extensive use of the open source ARToolkit libraries for raw fidicual tracking. As this data is generally quite noisy and spurious (especially when tracking many fiducials), we filter the data in several ways before sending it up to the simulator.

Since the ar_pose ROS node responsible for the tracking yields spurious results in which fiducials disappear/reappear from successive measurements, we implement a Bayes filter to estimate the visibility of the block. The filter uses an HMM (Hidden Markov Model) approach to estimate whether the block is present, intermittent, or not detected in the scene. This information is useful for manipulation tasks in the

simulator.

Additionally, a ROS node known as the Kalman fusion center is implemented that fuses the estimates of each of the fiducials, taking into account a slow dynamics model and movement. This allows multiple noisy cameras to be combined into a single, more accurate measurement.

4.4.4 World State Estimator

The World State Estimator is a hybrid state estimation module responsible for converting the continuous numerical pose estimates returned from the Kalman fusion center into a set of instantiated PDDL predicates describing the state of the world. This is essentially what implements the $\mathcal{W}_{obs}(t)$ in our formalisms presented in earlier chapters, and is they cornerstone of an execution monitoring system. It converts the continuous sensor measurements to state estimates at the plan level, namely PDDL predicates. The World State Estimator uses a variety of heuristics to do this, many of which are geometrically-inspired.

4.4.5 Executive

The executive is the core autonomy in the system. We have described the execution monitoring and generative planning aspects of the executive in detail in earlier chapters of this thesis. We note that for our robotic testbed, the TBurton executive was not yet functional. As such, we used an executive similar in spirit, but replacing the TBurton planning algorithm with an off the shelf planner - namely the Fast Forward planning system ([15]).

4.4.6 Activity Dispatcher

The Activity Dispatcher receives commands that have been dispatched by Pike, and further processes them so that they can actually be executed. It elaborates the often high-level commands from Pike to a lower level of abstraction. Oftentimes this involves calling a motion-planning algorithm using libraries provided by the OpenRAVE

simulator.

The Activity Dispatcher is capable of processing and executing the following PDDL actions:

- `(pick-up` *`Robot BlockTop BlockBase`*`)`
- `(pick-up-from-ground` *`Robot Block`*`)`
- `(stack-block` *`Robot Block BlockBase`*`)`
- `(put-block-on-ground` *`Robot Block`*`)`
- `(put-block-on-cart` *`Robot Block Cart`*`)`
- `(open-gripper` *`Robot`*`)`
- `(close-gripper` *`Robot`*`)`

Many of these PDDL actions, such as the top 5, require a number of different actions using the OpenRAVE simulator. For example, when `(pick-up-from-ground` *`Robot Block`*`)` is dispatched from Pike, the Activity Dispatcher is responsible for determining an acceptable grasping position for *`Block`* (our implementations seeks to find a valid solution to the inverse kinematics problem for several known reliable grasps of the block). Once this is complete, a valid motion to the block must be computed - we use the BiRRT algorithm implemented in OpenRAVE. It is thus the Activity Dispatcher's job to elaborate high-level actions from Pike into lower-level subroutines implemented by the Simulator.

4.4.7 Speech System

The speech system is responsible for both listening and processing human speech, and also for converting text from the executive into verbal speech. The speech module has a dictionary of known commands that the user can utter. We use the open source Sphinx libraries developed at Carnegie Mellon University for the text-to-speech and speech-to-text conversions.

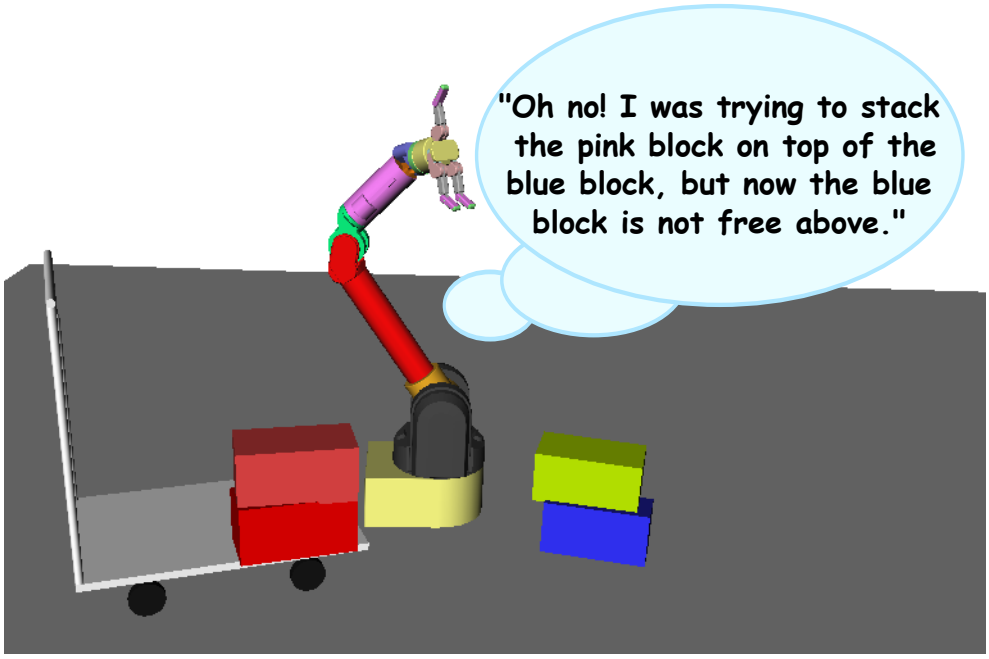

Figure 4-5: This figure illustrates output from our simulation environment when explaining the cause of a failure for an action. Suppose that while attempting to stack the pink block on the blue block, a malicious user moved the green block on top of the blue block. The spoken explanation is a direct verbalization of an violated causal link, indicating the failed action and violated predicate.

4.4.8 Explanation System

We mentioned earlier that commanding robots from a high level, and subsequently having them reason and respond at an equally high level, is a hallmark of artificially intelligent systems. In this section, we attempt to make strides towards that goal. Specifically, we attempt to verbalize the cause of failure when a causal link is violated to give the user an intuitive explanation of what went wrong.

Suppose that during execution, the execution monitoring subsystem of Pike detects that an activated causal link has been violated. This indicates that the precondition or maintenance condition of some future action will not hold true. The causal link provides information as to what action that is and what precondition will be violated, and what action was supposed to have produced that precondition. We argue that these three pieces can form the core of a high level explanation to the user,

describing what went wrong.

We implemented an algorithm that verbalizes violated causal links by converting them to textual representation. In essence, the actions a_p and a_c are mapped to strings (ex., "stack pink block on top of blue block"), as are the predicates ("blue block is free above"). Simple explanation can then be attained by concatening these string together to form an English sentence. For example, a violated causal link could be verbalized as "Oh no! I was trying to stack the pink block on top of the blue block, but now the blue block is not free above." Please see Figure 4-5 for an illustration. This explanation is a simplistic extension to the system, but demonstrates the richness of high-level information that can be gleaned from the causal link analysis of a plan.

4.5 Results

We were able to successfully construct a robot capable of accepting high-level input from the user, generating and executing those plans in the face of disturbances, and verbalizing sources of problems to the user at a high level - all within the limited context of a block stacking robot. This system demonstrates the efficacy of these algorithms on real robotic systems. In practice the robot was able to succeed at accomplishing its goals even as we bullied the poor robot by incessantly moving blocks around to jeopardize its current plan.

4.6 Chapter Summary

In this chapter, we have sketched a vision for artificially intelligent robotic systems, and proceeded to demonstrate in hardware and software capable of meeting some aspects of this vision within a limited context. We feel that expressive nature of causal links makes them an excellent tool for inferring the subgoals in a plan, and also provide useful information that can be communicated to the user at a high level.

Chapter 5

Conclusions

This thesis has revolved around the development of robust plan executives that operate in dynamic and highly uncertain worlds. We introduced a plan executive called Pike, capable of executing plans and simultaneously efficiently monitoring the progress with respect to possible problems for future actions. A core novel contribution of this thesis is the set of algorithms for extracting sets of candidate causal links from temporally flexible plans. These causal links are then monitored online. We prove guarantees of correctness for our algorithms, and additionally verify them experimentally with simulation results.

We also presented the TBurton Executive, a system combining the merits of temporal generative planning with the Pike plan executive. This system is capable of meeting a user's high-level goals, employing replanning to ameliorate problems that arise during execution.

The concept of generative planning combined with execution monitoring was demonstrated in hardware and in simulation within the context of a simplified manufacturing robot. This robot additionally employed the causal link information gleaned from the execution monitor to verbalize the reason why any relevant disturbances threatened the future of the plan.

Taken all together, this thesis strives to advance the state of robust robotic task execution. We developed new algorithms, prove theoretical guarantees, and empirically validated them on an engaging robotic testbed and simulation environment.

5.1 Interesting Observations

We noted many interesting observations throughout the research process of developing this system. It is interesting that some of these observations came about due to theoretical reasons as we proved guarantees about our algorithms, while others were observed empirically on our hardware testbed. This is an interesting testament to the necessity of doing both theoretical and applied work to advance state of the art.

A theoretically-motivated observation that was at first unexpected is the fact that, given a temporal plan, there is in general not a unique set of causal links that can be extracted from the plan. Due to the flexible durations within temporal plans, it may be possible for some actions to come before or after other actions, for different consistent schedules of the same plan. This greatly complicated the causal link process. Hence our causal link extraction algorithms are significantly more intricate than what we would have expected when we first began this research.

A second interesting observation, and possible topic for future work, is the notion that at times, if the hybrid state estimation module that outputs world state estimates is not completely correct, it can destabilize the executive. The TBurton executive is susceptible to getting caught in infinite loops of the state estimation fails to return a key fact, causing the robot to react by repeatedly trying to perform the same failing action. We propose that it would be interesting further research to better characterize the situations in which this interplay between the execution monitor and world state estimator may destabilize the system.

5.2 Future Work

In this section, we describe a small subset of the very broad topics that can be examined for future work.

Possible extension? They compliment our work. A system is envisioned in which the causal link extraction algorithms in this thesis can be generalized to extract the TAL monitorling coniditions.

5.2.1 Extension with Probabilistic Flow Tubes

Probabilistic Flow Tubes (PFT's) allow continuous actions to be learned by demonstration, and could provide an interesting extension to this research ([9]). Not only would it be interesting to examine the problem of learning new PDDL actions or motions via user demonstration and then subsequently applying them to new plans autonomously, but PFT's also present an interesting probabilistic method of evaluating flow tubes or performing intent recognition for execution monitoring. This idea could be further generalized to the notion of combining the hybrid estimation system with the execution monitor in order to provide more guarantees or a probabilistic estimate of failure.

5.2.2 Intent Recognition

We believe that the problem of extracting causal links from temporally flexible plans is closely related to the task of intent recognition, or namely, the idea of back-inferring what the users goals were from a sequence of generated actions (or even just an execution trace). This has significant implications for the future or robot human interaction, and when combined with the probabilistic flow tubes above, may provide a very interesting area of research for a Ph.D thesis.

5.2.3 Examine a Probabilistic Model of Disturbances

Closely related to the above is the challenge of utilizing a probabilistic model of disturbances. In this thesis, we make no guarantees about when or what disturbances will occur. However, in practice, robotic systems may be able to predict what disturbances are most likely to occur. For example, we noticed in our hardware testbed simulations that certain actions, such as picking up a block, were much more likely to fail than other actions (putting an already-held block on the ground). Therefore, it would be interesting to extend the notion of execution monitoring into the probabilistic domain. A key challenge would likely be to simplify the set of assumptions reasonably enough so that the problem is still tractable and monitoring can be per-

formed in real time, while simultaneously keeping a rich representation of possible disturbances. Such a system has deep ties with the intent recognition ideas above, and when combined, is an area of research we are excited about and considering pursuing.

5.2.4 Advancements in Dialog System

The dialog system employed in this testbed was relatively simplistic. Causal links were directly verbalized by mating predicates and actions to their string representations. It would be very interesting however to leverage more advanced natural language and dialog systems in this research. Many argue that language is one of the most fundamental facets of human understanding, and hence we may be able to build better simulations of intelligent systems by employing a more advanced dialog system with our causal link explanation approach.

Bibliography

[1] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 264. IEEE, 2004.

[2] D. Colbry, B. Peintner, and M.E. Pollack. Execution monitoring with quantitative temporal bayesian networks. *Ann Arbor*, 1001:48103, 2002.

[3] P.R. Conrad and B.C. Williams. Drake: An efficient executive for temporal plans with choice. *Journal of Artificial Intelligence Research*, 42:607–659, 2011.

[4] J. De Kleer and B.C. Williams. Diagnosing multiple faults. *Artificial intelligence*, 32(1):97–130, 1987.

[5] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1):61–95, 1991.

[6] R. Diankov and J. Kuffner. Openrave: A planning architecture for autonomous robotics. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34*, 2008.

[7] Rosen Diankov. *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.

[8] P. Doherty, J. Kvarnström, and F. Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

[9] S. Dong and B. Williams. Motion learning in variable environments using probabilistic flow tubes. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1976–1981. IEEE, 2011.

[10] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 3:251–288, 1972.

[11] R.E. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1972.

[12] R.W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[13] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[14] C. Fritz and S.A. McIlraith. Monitoring plan optimality during execution. 2007.

[15] J. Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3):57, 2001.

[16] A.G. Hofmann. *Robust execution of bipedal walking tasks from biomechanical principles*. PhD thesis, Massachusetts Institute of Technology, 2005.

[17] P. Kim, B.C. Williams, and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 487–493. LAWRENCE ERLBAUM ASSOCIATES LTD, 2001.

[18] J. Kvarnström, F. Heintz, and P. Doherty. A temporal logic-based planning and execution monitoring system. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS), Sydney, Australia*, 2008.

[19] S. Lemai and F. Ingrand. Interleaving temporal planning and execution in robotics domains. In *Proceedings of the National Conference on Artificial Intelligence*, pages 617–622. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2004.

[20] D. McAllester and D. Rosenblatt. Systematic nonlinear planning. 1991.

[21] C. Muise, S.A. McIlraith, and J.C. Beck. Monitoring the execution of partial-order plans via regression. In *Proc. of 22nd International Joint Conference on Artificial Intelligence*, pages 1975–1982, 2011.

[22] N. Muscettola, P. Morris, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *In Principles of Knowledge Representation and Reasoning*. Citeseer, 1998.

[23] N. Muscettola, P.P. Nayak, B. Pell, and B.C. Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[24] J.S. Penberthy and D. Weld. Ucpop: A sound, complete, partial order planner for adl. In *proceedings of the third international conference on knowledge representation and reasoning*, pages 103–114. Citeseer, 1992.

[25] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 1995.

[26] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 2010.

[27] M. Soutchanski. Execution monitoring of high–level temporal programs. In *Robot Action Planning, Proceedings of the IJCAI-99 Workshop*, pages 47–54. Citeseer, 1999.

[28] A. Tate. Generating project networks. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2*, pages 888–893. Morgan Kaufmann Publishers Inc., 1977.

[29] M.M. Veloso, M.E. Pollack, and M.T. Cox. Rationale-based monitoring for planning in dynamic environments. In *Proceedings of the Fourth International Conference on AI Planning Systems*, pages 171–179, 1998.

[30] B.C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *International Joint Conference on Artificial Intelligence*, volume 15, pages 1178–1185. LAWRENCE ERLBAUM ASSOCIATES LTD, 1997.

[31] B.C. Williams and R.J. Ragno. Conflict-directed a* and its role in model-based embedded systems. In *Journal of Discrete Applied Mathematics*. Citeseer, 2003.